POLITECNICO DI TORINO
Repository ISTITUZIONALE

Rethinking Software Network Data Planes in the Era of Microservices

Original
Rethinking Software Network Data Planes in the Era of Microservices / Miano, Sebastiano. - (2020 Jul 13), pp. 1-175.

Availability:
This version is available at: 11583/2841176 since: 2020-07-22T19:49:25Z

Publisher:
Politecnico di Torino

Published
DOI:

Terms of use:
Altro tipo di accesso

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

05 May 2023

Doctoral Dissertation
Doctoral Program in Computer and Control Enginering (32nd cycle)

Rethinking Software Network Data Planes in the Era of Microservices

Sebastiano Miano
* * * * * *

Supervisor
Prof. Fulvio Risso

Doctoral examination committee
Prof. Antonio Barbalace, Referee, University of Edinburgh (UK)
Prof. Costin Raiciu, Referee, Universitatea Politehnica Bucuresti (RO)
Prof. Giuseppe Bianchi, University of Rome "Tor Vergata" (IT)
Prof. Marco Chiesa, KTH Royal Institute of Technology (SE)
Prof. Riccardo Sisto, Polytechnic University of Turin (IT)

Politecnico di Torino
2020

This thesis is licensed under a Creative Commons License, Attribution - Noncommercial-NoDerivative Works 4.0 International: see www.creativecommons.org. The text may be reproduced for non-commercial purposes, provided that credit is given to the original author.

I hereby declare that, the contents and organisation of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

. . . . . .

Sebastiano Miano

Turin, 2020

Summary

With the advent of Software Defined Networks (SDN) and Network Functions Virtualization (NFV), software started playing a crucial role in the computer network architectures, with the end-hosts representing natural enforcement points for core network functionalities that go beyond simple switching and routing. Recently, there has been a definite shift in the paradigms used to develop and deploy server applications in favor of microservices, which has also brought a visible change in the type and requirements of network functionalities deployed across the data center. Network applications should be able to continuously adapt to the runtime behavior of cloud-native applications, which might regularly change or be scheduled by an orchestrator, or easily interact with existing "native" applications by leveraging kernel functionalities - all of this without sacrificing performance or flexibility.

In this dissertation, we explore the design space of software packet processing applications within the new "cloud-native" era, and we propose a novel paradigm to design, run, and manage software network functions that follow the same approach of micro-services. We present Polycube, a software framework that enables the creation of efficient, modular, and dynamically reconfigurable in-kernel networking components available with vanilla Linux. Polycube exploits the extended Berkeley Packet Filter (eBPF) framework to execute the data plane of those network functions and introduces a set of additional components and common APIs that make it easier to develop and manage those services. We design and evaluate the use of this paradigm through `bpf-iptables`, a clone of `iptables` characterized by improved performance and scalability. Then, we explore the possibility of enhancing the capabilities of end-hosts through the use of programmable network interface cards (SmartNICs) to offload partially (of fully) existing packet processing applications, in particular in the domain of DDoS Mitigation. In the last part of the dissertation, we present Kecleon, a compiler framework that can be used to dynamically optimize generic software data planes, taking into account the runtime characteristics and packet processing behavior of the original network function. We believe that the combination of these works can lay the foundation for a new model of packet processing applications that is better suited for modern cloud environments, having the capability to be dynamically re-combined, re-generated, and re-optimized without sacrificing programmability, extensibility and performance.

Acknowledgements

This Ph.D. has totally changed my personality and way of thinking, thanks to all the wonderful (and, sometimes bad) experiences that I got and the amazing people that I met and worked with. It was a journey that I would always remember and something that I will definitely start again if I could. Many people have assisted me during this journey and I would like to express my gratitude to all of them for helping me become a researcher and the person that I am today.

First of all, I would like to thanks my advisor, Prof. Fulvio Risso for all the lessons that I learned from him, for his patience, his immense love in the field, and his capacity to drive me in the right directions when it was necessary. I will be always grateful to him.

My colleagues and friends Matteo Bertrone and Mauricio Vasquez (the initial Polycube team), with whom I have shared all the years of my Ph.D. and who helped me to refine my research and provided invaluable help when I needed it most. It is also thanks to them that I was able to reach a certain level of maturity and robustness of the works presented in this thesis.

All the people and friends that I have met in Cambridge. Prof. Andrew Moore for giving me the chance to do this wonderful experience, and Christos, Hilda, Salvador, Yuta, and Marcin for all the interesting discussions that we had and for helping me (a poor Sicilian guy) to survive in the UK during the entire summer. A huge thanks go to Prof. Gianni Antichi, from having welcomed me in Cambridge and for teaching me a lot in the last year. He became a real role model for me and a great friend.

A great thanks to my parents that gave me the change to do a Ph.D. and bring me up to love school and science, and that always supported me during the entire Ph.D. A special thanks to my sister, Debora for all the suggestions and support that she gave me and for being always present when needed. She was the first person that I always called when, in panic, I was looking for advice.

Last but not least, my biggest thanks to Valentina. Although sometimes I neglected her to focus on the Ph.D., she has been always on my side and supported me in any choice I have made, even if this sometimes took me away from her.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

One of the main reasons behind the success of Internet and its widespread revolves around the "end-to-end" principle [138, 15]. The general idea behind it is that, a network designed according to this principle should keep application-specific features on the end-hosts of the network while maintaining its *core* simple. Intermediate nodes, such as switches and routers, are just "dumb" forwarding devices that send packets between one end-host to another, while more sophisticated features are implemented in the software running on the end-host devices.

During the years, this approach has been successfully applied to both wide area networks (WAN) and data center networks (DCN). The simplicity and stateless model of the *core* network have facilitated the implementation of fixed functionalities running on specialized network hardware processors or application-specific integrated circuit (ASIC), fostering the introduction of more network providers that led to the rapid growth of the Internet traffic. On the end-hosts, the increased flexibility and programmability of the software have encouraged the innovation resulting in the realization of ground-breaking applications such as the World Wide Web.

While the end-to-end principle still stands as a fundamental design framework in computer networking, in recent years, the distinction between the "dumb core" and the "smart edge" has gotten blurred, with different trends that tip the balance more on a direction or the other. Modern network providers (ISPs) started running their networks as a business, relying on the provision of value-added services such as content distribution [137, 6, 122], safe browsing, or anti-malware, also rented to third-parties to provide more advanced features and increase their revenue [95]. Data center networks followed the same path; the growing need for security and reliability has caused the network to implement more sophisticated features such as proxies, intrusion detection and prevention systems, firewalls, caches and load-balancers, implemented within dedicated hardware network appliances (i.e., middleboxes) built for the specific application purposes [140].

On the other hand, motivated by the high infrastructure and management costs, which result from the complex and specialized processing of the above-mentioned

devices, the advent of "network softwarization" proposed traditional network service to be transformed into pure software images that can be executed on (cheap) general-purpose servers - running on VMs or cloud sites. Then, by using Software Defined Networking (SDN) [131], network traffic can be steered through a chain of Virtual Network Functions (VNFs) in order to provide aggregated services.

With the introduction of these concepts, it becomes more evident that *software* started playing a crucial role in the network dataplane architectures, with the end-hosts that increasingly became a natural enforcement point for core network functions such as load balancing [76, 94, 74, 16], congestion control [3, 89, 119], and application-specific network workloads - thanks also to the development and availability of programmable network devices (e.g., SmartNICs). The new requirements in terms of flexibility (software is intrinsically easier to program and to update compared to the hardware implementations), and the recent advances in terms of speed for the software packet processing have contributed to the proliferation of a myriad of VNFs frameworks that provide implementations of efficient and easily programmable software middleboxes [35, 90, 169, 86, 93, 100, 170, 123, 125].

Current solutions to implement the dataplane of those *software packet processing* applications rely mostly on kernel bypass approaches, for example by giving to the user-space direct access to the underlying hardware (e.g., DPDK [54], netmap [133], FD.io [83]) or by following a *unikernel* approach, where the only the minimal set of OS functionalities, required for the application to run, are built within the application itself (e.g., ClickOS [106, 66, 96]). These approaches have perfectly served their purposes, with efficient implementations of software network functions that have shown potential for processing 10-100Gbps on a single server [134, 105, 72].

Recently, the advent of new networking technologies such as 5G, edge computing, IoT, has brought a significant increase in the total number of connected devices and cloud service load, requiring network operators to change the previously monolithic paradigm used to develop and deploy server applications in favor of *micro-services*. Cloud-native technologies are used to develop applications built with services packaged in containers, deployed as microservices, and managed on elastic infrastructure through agile DevOps processes and continuous delivery workflows [87]. This paradigm shift has also brought a visible change in the type and requirements of network functionalities deployed across the data center, given the new workloads and applications running on the servers. For instance, cloud-native platforms, like Kubernetes [82], rely on different network providers (a.k.a., network plugins) to implement the underlying data plane functionalities and transparently steer packets between the micro-services.
Those new requirements can be summarized by the following points:

- **Runtime flexibility:** A single cloud-native application is composed of several services, each of which has thousands of instances that might constantly

change and be dynamically scheduled by a data center software called orchestrator. Within this world, the service communication is fundamental, and the network should follow the same behavior by continuously adapting to the runtime characteristics to ensure end-to-end performance and reliability [13].

- **Low overhead:** Networking components should handle the communication of each server in the cluster. As a consequence, the cost to be paid for running network functions, in terms of resource consumption, for each server becomes significant. This is even more evident within edge clouds, where the number of available resources is limited.

- **Agile service development:** One of the biggest advantages of the microservice paradigm, together with the concept of building loosely-coupled, fine-grained services is the *continuous delivery* (CD) software development process; a small change to the application requires re-building and re-deploying only a small part of the entire service. Newer software data planes should follow the same approach, making it possible to easily update the existing application by providing a replacement that does not disrupt the typical service workflow [43].

- **Automatic optimizations:** From a developer's point of view, writing an efficient and, at the same time, easy to maintain software data plane is a daunting task. Most of the time is just a matter of finding the right trade-off between *simplicity* (e.g., modularity, easy-to-read code) and the *performance*. As a consequence, we often see application-specific, and ad-hoc techniques applied only to particular use-cases [72, 105], which do not perform well on other scenarios or for a broader spectrum of applications. To better adapt to this new "frenetic" environment, the network components should be able to automatically adapt themself to the runtime condition with the minimum amount of programming overhead.

- **Work with vanilla Linux kernel:** Applications are now running on the host operating system that is shared between the different components (e.g., containers), which in turn rely on existing kernel functionality to accomplish their tasks [9]. It is then crucial that network functions can easily interact with existing "native" applications and also leverage kernel functionalities (e.g., TSO, skb metadata, etc.), without sacrificing performance and flexibility.

Unfortunately, the previously mentioned kernel-bypass approaches are at a stake when adopted in this new scenario, for different reasons [120]. First, they require the exclusive allocation of resources (i.e., CPU cores) to achieve good performance; this is perfectly fine when we have a single dedicated machine for the networking purposes but it becomes overwhelming when this cost has to be paid for every server in the cluster since they permanently steal precious CPU cycles to other application

tasks. Second, they require to re-implement the entire network stack in userspace, losing all the well-tested configuration, deployment and management tools developed over the years within the operating system. Third, they rely on custom or modified versions of network drivers, which may not be available on on-demand cloud platforms, also requiring a non-negligible maintenance cost. Last but not least, they have difficulties (and poor performance) when dealing with existing kernel implementations or communicating with applications that are not implemented using the same approach, requiring them to adhere to custom-defined APIs (e.g., mTCP [88]) or to change the original application logic (e.g., StackMap [165]). As a consequence, most of the existing cloud-native network providers today still rely on functionalities and tools embedded into the operating system network stack (e.g., iptables, ipvs, linux bridge). Unfortunately, the drawbacks of this approach are also evident. First of all, *fixed* kernel network applications are notoriously slow and inefficient given their generality, which impairs the possibility to specialize the software network function depending on workloads or the type of application that is running on top of it. Secondly, software network functions that live in the kernel have also proven hard to evolve due the complexity of the code and the difficulties in maintaining, up-streaming or modifying the kernel code (or the respective kernel modules).

In this dissertation, we present our steps towards the definition and realization of a novel class of software network applications that can fulfill those new requirements, removing the limitation seen with the existing approaches. We first present **Polycube**, a framework that can be used to build flexible and efficient in-kernel software network functions that follow the same approach of "cloud-native" applications, enabling the creation of efficient, modular and dynamically reconfigurable networking components, available with vanilla Linux. It exploits the recently added eBPF [107] subsystem to build the data plane of the network function. We will show how Polycube enables the writing of complex networking components that can be used to provide flexible and optimized replacements of existing (fixed) in-kernel implementations such as *iptables*, while keeping the same semantic and syntax of the original application, but with improved performance. Then, we move the focus to **Kecleon**, a compiler framework that can be used to dynamically optimize software data planes at run time by adopting a dynamic approach to the data path compilation, where not only the static features but also the runtime data of the original application are exploited to generate a custom version of the original data plane that would be optimal to the data plane semantic and the packet processing behavior at the same time. While the two systems focus on different layers, the combination of both can lay the foundation of a new paradigm in building software packet processing applications that can be dynamically re-combined, re-generated, re-compiled and re-optimized without sacrificing performance, programmability or extensibility.

1.1 Summary of Contributions

We start the first part of the dissertation (Chapter 3) by exploring the possibility of using the extended Berkeley Packet Filter (eBPF) [107] as a base subsystem to build *complex* software packet processing applications. So far, eBPF was mainly used for monitoring tasks such as memory, CPU, page faults, traffic, and more, with a few examples of traditional network services, e.g., that modify the data in transit. However, it also has some characteristics that make it the right candidate for running data plane applications. First of all, eBPF can process a packet entirely in kernel space, without context switches or packet copies between kernel and user space. Second, it leverages a set of features that are already present in a modern Linux kernel, without requiring additional kernel modules that are difficult to create and maintain. Third, the possibility to compile and inject the code at runtime paves the way to context-based customization of each network function. Eventually, eBPF programs can cooperate with the kernel TCP/IP stack, possibly complementing existing networking features. We make the following contributions. We first analyze the possibility to create complex network functions that go beyond simple proof-of-concept data plane applications, and we present the most promising characteristics of this technology. Then, we indicate the main encountered limitations, and some solutions that can mitigate the latter. Second, we summarize the most important lessons and, finally, we provide a quantitative characterization of the most significant aspects of this technology.

The above findings made us reaching to the conclusion that creating network functions based entirely on eBPF is sometimes complicated given the lack of a common framework that provides useful abstractions to developers to solve common problems or known limitations. Even though eBPF allows complex and user-defined operations to be performed in the kernel, it is not Turing-complete. As a consequence, it cannot support truly arbitrary processing, making the implementation of some common functionalities (e.g., ARP handling in a router) challenging. Furthermore, no abstractions current exist to implement the (complex) control plane of a service, hence forcing developers to dedicate a considerable amount of time to handle common control plane operations (e.g., user-kernel interaction). This motivated us to rethink how to enable a network function model where in-kernel network applications can be adjusted, injected and modified in a simple and defined way, hence enabling the dynamicity and flexibility required by new data center workloads. We present Polycube (Chapter 4), an overarching coherent software architecture that allows *in-kernel* network services to be managed in a logically centralized manner. With Polycube, we make the following contributions. First, we make it possible to create complex network services that can overcome the eBPF limitations. We define a specific structure of a service where each function can include an efficient in-kernel data plane (based on eBPF) and a flexible user-space control plane. Then,

we handle through a specific programming API the interaction between the different components, while keeping strong characteristics of isolation and composability. Second, we manage the creation of complex network applications by allowing the arbitrary construction and concatenation of different services to create arbitrary chains of in-kernel network functions. Third, we introduce a generic model for the control and management plane of each network function that simplifies manageability and accelerates the development of new network services. We make it possible to change the current version of a given Polycube service dynamically or to modify the service chain at runtime, hence achieving a new type of customizability and flexibility inside the Linux kernel. Finally, we identify and quantify the overhead introduced by the Polycube abstractions, and we show the design and the performance improvements that Polycube services can bring into the new cloud-native environment (e.g., by presenting an implementation of a k8s CNI plugin). In a nutshell, Polycube allows us to achieve the same modularity, flexibility, and development process that is possible with user-space Network Functions but inside the Linux kernel, with all the benefits that go with it.

To better present the advantages of this new model and the power of Polycube, in Chapter 5 we present `bpf-iptables`, a fully-functional replacement of `iptables` that emulates its filtering semantic but exploiting a more efficient matching algorithm. We show how `bpf-iptables` achieves a notable boost in terms of performance compared to the current implementation of iptables, particularly when a high number of rules is involved, all of this within a vanilla Linux kernel. We make three main contributions in this work. Firstly, we present the design of `bpf-iptables` together with the main challenges and possible solutions to preserve the original `iptables` filtering semantic. To the best of our knowledge, `bpf-iptables` is the first solution that provides an implementation of the `iptables` filtering completely in eBPF. Secondly, we give a comprehensive analysis of the main limitations and challenges required to implement a fast matching algorithm in eBPF, keeping into account the current limitations of the above technology. Third, we show a set of data plane optimizations that are possible thanks to the flexibility and dynamic compilation (and injection) features of eBPF, allowing us to create at runtime an optimized data path that fits perfectly with the current ruleset being used.

To further enhance the capabilities of end hosts, while keeping the best possible trade-off between *resource consumption* and *performance*, we have evaluated the possibility to exploit *programmable* network interface cards (a.k.a., SmartNICs) to offload partially (or fully) existing packet processing functions. In particular, we took the DDoS mitigation as a use-case for this study. In Chapter 6, we then present an architecture that can be used to transparently offload a portion of DDoS mitigation rules into a SmartNIC, thus achieving a balanced combination of the in-kernel packet processing flexibility with eBPF to operate traffic sampling and aggregation, with the performance of hardware-based filtering. We first analyze the various approaches that can be used to design an efficient and cost-effective DDoS mitigation

solution. As generally expected, our results show that offloading the mitigation task to the programmable NIC yields significant performance improvements. However, we also demonstrate that due to the memory and compute limitations of current SmartNIC technologies, a fully offloaded solution may lead to deleterious performance. Second, as a consequence of the previous findings, we propose the design and implementation of a hybrid mitigation pipeline architecture that leverages the flexibility of eBPF/XDP to handle different types of traffic, and the efficiency of the hardware-based filtering in the SmartNIC to discard traffic from malicious sources. Third, we present a mechanism to transparently offload part of the DDoS mitigation rules into the SmartNIC, which takes into account the most aggressive sources, i.e., the ones that primarily impact on the mitigation effectiveness.

Starting from the experience gained in the design and implementation of this new type of in-kernel network services (e.g., `bpf-iptables`), we noticed that one of the most important characteristics that contributed to the better efficiency and performance of these applications is the possibility to *specialize* the corresponding network data plane *at runtime*, according to the specific application logic. For example, in `bpf-iptables` a lot of optimizations are applied at runtime according to the type of organization of the rules in the data set; this, combined with the dynamic injection of eBPF program in the kernel allowed us to notably outperforming all the other existing kernel implementations. Motivated by this work, we then tried to answer the following question; "is it possible to perform these runtime optimizations outside of the application context?". Taking a step back, we realized that traditional approaches to design and develop packet processing functions are based on a static compilation. The compiler's input is a description of the forwarding plane semantic, and the output is a binary code that can accommodate any pre-defined processing behavior. Although improving hot-code paths during the execution of a software program is nowadays possible with compilers that support Profile Guided Optimizations (PGO), Feedback Directed Optimizations (FDO), those techniques have been architected to optimize generic computer programs. Therefore, they do not easily accommodate the requirements of network data planes programs that have packets as input. In the last part of this dissertation, we then present Kecleon, a runtime compiler for generic packet processing applications that can dynamically generate an optimized version of the original application's data plane depending on its runtime behavior. Kecleon is designed to be independent not only from the application itself but also from the technology on which the application is implemented. It applies several dynamic optimizations at the compiler's Intermediate Representation (IR) level, to either be generic and to also transparently exploit existing compiler's analysis techniques and optimizations. Kecleon takes into account *(i)* the network configuration, e.g., to prune branches and instructions that are considered unreachable at runtime, *(ii)* the run time table content, e.g., selecting the most appropriate data structure (hash-based versus tree) to store the current data can speed up the lookup process and *(iii)* traffic patterns,

e.g., detecting the most frequently accessed entries to create an optimized fast-path code for them. We illustrate the details of Kecleon in Chapter 7.

1.2 Outline

The rest of this dissertation is organized as follows. Chapter 3 briefly introduces the eBPF subsystem and its main features, together with several insights and limitations we have found while implementing complex data plane applications with this technology. Chapter 4 presents Polycube, a software architecture that applies the micro-service paradigm to the world of in-kernel network functions. Chapter 5 shows one of its applications, `bpf-iptables`; a more efficient and scalable clone of `iptables` built around eBPF. Then, Chapter 6 focuses on the DDoS mitigation use case, and presents a study of the combination of the in-kernel packet processing flexibility performed with eBPF with the performance of programmable hardware-based filtering performed into a SmartNIC. In Chapter 7, we present Kecleon, a compiler framework that can be used to dynamically optimize software data planes at run time. Finally, Chapter 8 concludes this dissertation and discusses suggestions for future work.

Previously Published Work. This thesis includes previously published and co-authored works. In particular, Chapter 3 is adapted from [114], Chapter 4 from [108] and Chapter 5 from [115, 22]. Chapter 6 is adopted from [109], which is the result of a collaboration with the research center "Fondazione Bruno Kessler: FBK". Finally, the work in Chapter 7, which is still in a preliminary phase, is the result of a collaboration with the University of Cambridge, UK, and it has not been published yet.

1.3 Research Projects Not Included in This Dissertation

As part of the master course, the author worked on other topics that are not covered in this dissertation. However, they fall under the common goal of building more efficient and flexible networks at the edge of the network (e.g., on resource-contrained Customer Premise Equipment). Even though these topics were explored almost three years ago, their motivations and results achieved are still considered timely given the recent works that go on the same directions (e.g., [139], [62], [40]).

- **Transforming a Traditional Home Gateway into a Hardware accelerated OpenFlow switch** [112, 113]. Software Defined Networking (SDN) proposes a new paradigm that allows network administrators to manage network services from a centralized point of control through abstraction of lower

level functionality. The SDN innovation brought significant advancements to different areas, such as the administration of home networks. Traditional home gateways are, however, hard to manage as new application are introduced and moved at the customer premises; applying SDN to these devices would enable to program and control the home network from a centralize point of control, allowing users to manage and configure the behavior of their network via high-level applications.

In this work, we described our experience in porting OpenFlow on already existing hardware switch with no support for the OpenFlow standard. We presented our architecture that integrates a hybrid software and hardware pipeline and that is able to compensate the hardware limitations in terms of supported matches and actions, offloading only part of the OpenFlow rules, which are properly translated into the corresponding hardware related commands. We illustrated the design choices used to implement the functionalities required by the OpenFlow protocol (e.g., packet-in, packet-out messages) and finally, we evaluated the resulting architecture, showing the significant advantage in terms of performance that can be achieved by exploiting the underlying hardware, while maintaining an SDN-type ability to program and to instantiate desired network operations from a central controller.

- **Enabling NFV Services of Resource-Constrained CPEs** [27]. Virtual Network Functions (VNFs) are often implemented using virtual machines (VMs), since they provide an isolated environment compatible with classical cloud computing technologies. Unfortunately, VMs are demanding in terms of required resources and therefore are not suitable for resource constrained devices such as residential CPEs. Such hardware often runs a Linux-based operating system that supports several software modules (e.g., iptables) that can be used to implement network functions (e.g., a firewall), which can be exploited to provide some of the services offered by simple VNFs, but with reduced overhead.

  In this work, we proposed and validated an architecture that integrates native software components in a Network Function Virtualization (NFV) platform, making their use transparent from the user's point of view. Our solution enables an NFV orchestrator to optimize the scheduling of the Network Functions (NFs) by initialing services that require to be close to end users such as IPsec terminators or low-latency services, directly on the user CPE, while other components of the same service (e.g., the NAT module) are executed in a remote data center.

Chapter 2

Background and Motivations

In this chapter, we explore the available alternative that we have today to build software packet processing applications, along with their motivations, implications, and challenges. First, we explore the most important characteristics of packet processing performed in kernel-space or user-space, showing the corresponding pro and cons of both approaches (section 2.1). Finally, we provide a brief overview of the extended Berkeley Packet Filter subsystem (section 2.2) and why we think it may be beneficial for networking applications (section 2.2.1).

2.1 Userspace vs. Kernelspace networking

In recent years, thanks to the introduction and spread of Network Function Virtualization (NFV), software has gained a considerable importance, with a lot of network functionalities deployed at the end host that go beyond traditional switching and routing between VMs and containers. Current alternatives to building such software network applications can be organized into two categories: *user-level* (or kernel-bypass) approaches, and *kernel-level* implementations, where we use functionalities embedded in the operating system networking stack.

Userspace networking. Moving functionalities from hardware to software has increased performance demands for software network applications, favoring the appearance of specialized network facilities such as DPDK, netmap, and SRIOV. To avoid expensive user-kernel transitions, those systems directly access the network hardware from userspace, writing their network stack focused on performance and optimizations. As a results, they produce considerable benefits both in terms of throughput and latency. However, they have also some drawback, which we summarize within the following points:

- **High resource consumption.** Most of the kernel-bypass solutions require the exclusive allocations of one (or more) CPU cores to handle high-speed

packet processing. This model works well for replicating in software a "middlebox" approach, where a single machine is dedicated to networking purposes (e.g., routing, switching, or load-balancing). However, in a scenario (e.g., new cloud environments) where most of the machines (and CPUs) are dedicated to userspace applications (e.g., web servers), those approaches may result overwhelming.

- **Difficult integration with "native" applications/environments.** Applications that are currently relying on kernel-level functionalities or using kernel (e.g., socket) APIs cannot be transparently used when the userspace application has direct access to the underlying network hardware. Several companies have debugging tools developed over the years that cannot be used with kernel-bypass approaches [104]; Kubernetes network providers still count on *iptables*, *ipsets*, *ipvs* to provide security or load-balancing between containers. Solutions exist to allow their co-existence. For instance, the Kernel Native Interface (KNI) in DPDK or Netmap allow passing packets between the userspace application to the kernel network stack, or StackMap [165] that allows using the kernel TCP/IP stack directly. Unfortunately, these solutions will either result in sub-optimal performance [101] or will require the use of custom drivers, custom kernel modules, or modified kernels, which are not always available in every environment.

- **Security issues.** Bypassing the Linux networking stack does not only implies an incompatibility with several applications but also may bring security issues [60]. The Linux TCP/IP stack has some non-trivial and incredibly powerful features that have been developed over the years and that are well-tested and quite reliable. Of course, re-writing all from scratch will have the advantage of better design and then better performance [88, 105], but it would still require a lot of time before having a stable userspace networking stack comparable, in terms of features and compatibility, to the Linux kernel one.

Kernelspace networking. The alternative to handling all packets from a userspace process is to perform the processing entirely in the kernel, reducing at minimum the number of kernel/userspace transitions, which are the primary source of overhead. Except for the evident advantages of compatibility with existing applications and tools, and the ability to run within a vanilla Linux kernel, this approach also has some drawbacks.

- **Poor performance.** The Linux TCP/IP stack is made of several layers and abstractions given by the generality of the design, which has to be valid for several types of hardware and architecture. On one side, this offers better compatibility between the different layers and a good level of abstraction from

the underlying hardware, but on the other hand, it is also the main source of the poor performance of the system. A packet entering the Linux TCP/IP stack has to cross all the different layers, making it very hard to bypass or "jump" from one layer to another.

- **Difficult development process.** Adding kernel-level functionalities is a non-trivial and lengthy process. Writing custom kernel modules may require a non-negligible effort to keep compatibility across the different kernel versions, or it may incur reluctance from customers to include them in the production system (a crash in the kernel module will hurt the entire system). On the other hand, upstreaming functionalities into the mainline kernel requires to "convince" maintainers of their relevance for the whole of the community. It often results in either long delays before a feature is accepted and available in future kernel releases, and it would break up the innovation cycles for a company that wants to implement features that are useful only for their use cases.

Until a few years ago, the approaches mentioned above were the only alternatives available to develop kernel-level network functionalities. However, the recent introduction and evolution of the "classic" BPF [107] subsystem, called extended BPF, has introduced exciting features and concepts that make developing and running kernel-level functionalities easier and more efficient.

2.2 The extended Berkley Packet Filter (eBPF)

The Berkeley Packet Filter (BPF) is an in-kernel virtual machine for packet filtering that has been deeply revisited starting from 2013 and is now known as extended BPF (eBPF). In addition to several architectural improvements, eBPF introduces the capability of handling generic event processing in the kernel, JIT compiling for increased performance, stateful processing using maps, and libraries (helpers) to handle more complex tasks, available within the kernel.

eBPF allows a user-space application to inject code in the kernel at runtime, i.e., without recompiling the kernel or installing any optional kernel module. eBPF programs can be either written using eBPF assembly instructions and converted to bytecode using `bpf_asm` utility or in restricted C and compiled using the LLVM Clang compiler. The bytecode can then be loaded using the `bpf()` system call. For this process to succeed, the program has to get through a sanity-check from the eBPF verifier, that walks the control flow graph to ensure termination, simulates the execution to check that memory and registers are always in a valid state, and verifies that the calls to helper functions are allowed.

A loaded eBPF program follows an event-driven architecture and it is therefore hooked to a particular type of event (e.g., the arrival of a packet). Each occurrence

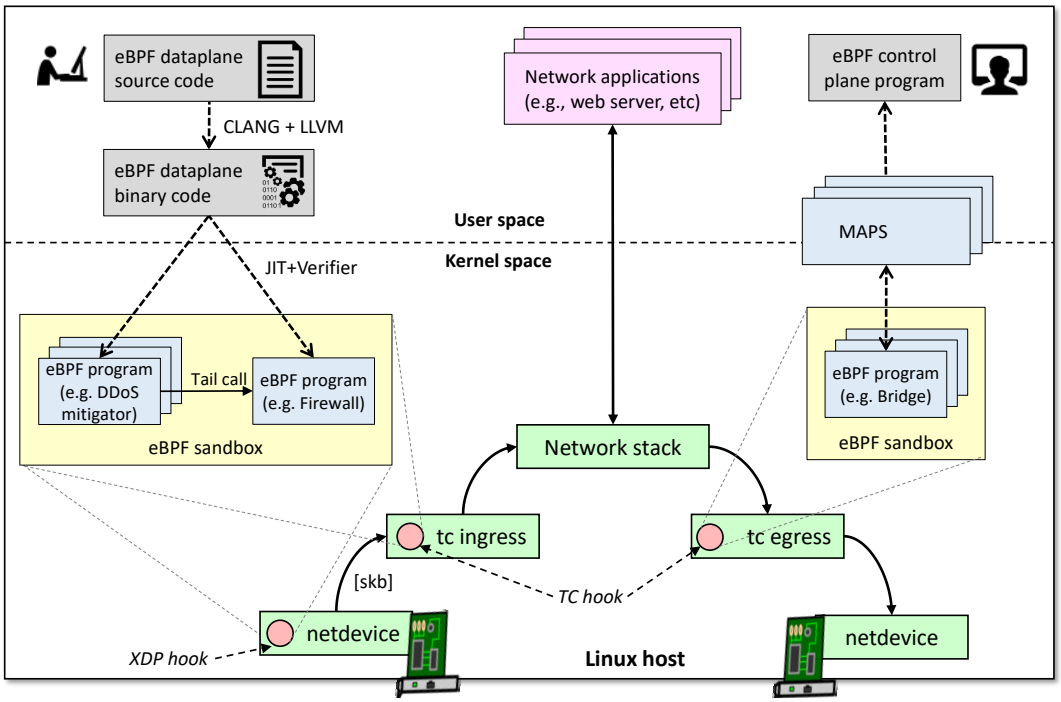
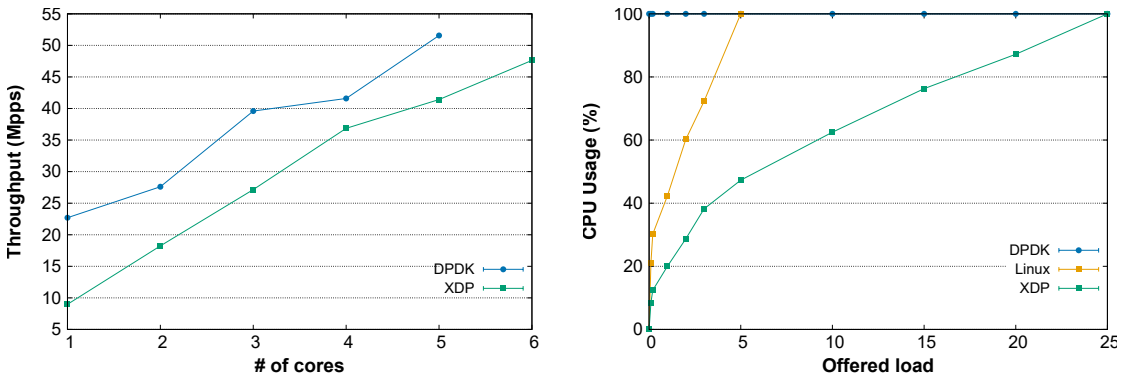


Figure 2.1: eBPF overview.

of the event will trigger the eBPF program execution, and, based on the type of event, the program might be able to alter the event context[1]. Furthermore, programs are stateless by their nature, as each run is independent of the others. For this reason, the eBPF provides *maps*, data structures accessible using helper functions, needed to share information between (*i*) different runs of the same program, (*ii*) various programs, or (*iii*) a program and the userspace.

For networking purposes, program execution is triggered by the arrival of a packet. Two hooks are available to intercept packets and possibly mangle, forward or drop them: eXpress Data Path (XDP) and Traffic Control (TC). XDP programs intercept RX packets right out of the NIC driver, possibly before the allocation of the Linux socket buffer (`skb`), allowing, e.g., early packet drop. TC programs intercept data when it reaches the kernel traffic control function, either in RX or TX mode. Multiple eBPF programs can be instantiated at the same time, even attached to different hooks. Furthermore, eBPF programs can either operate in isolation (returning the packet to the hook they are attached to) or be chained, e.g., to create a more complex service, using a low-overhead linking primitive called

[1]For networking program, the event context is represented by the packet itself, while for eBPF program attached to generic kernel function, the context is represented by the function's arguments.



(a) (b)

Figure 2.2: (a) Forwarding performance comparison between XDP and DPDK with small packets (64B) redirected between different NICs. DPDK uses one control thread, so only 5 to 6 are available for the forwarding. (b) CPU usage differences between DPDK, XDP, and Linux when dropping packet with a variable offered load. The data were obtained from [75].

tail call. Tail calls are a sort of long jump from one program to another; differently from *function calls*, this primitive does not permit to return to the previous context.

A high-level view of the eBPF architecture, including both code injection and run-time processing, is depicted in Figure 2.1.

2.2.1 eBPF for Network Functions

The most attractive features that we have found in the eBPF subsystem can be summarized within the following points:

- **Easy development process.** eBPF programs follow the same development process of userspace applications. Custom applications can be developed independently from the kernel development process. They can be dynamically injected into the kernel using the `bpf()` system call, without having to install custom kernel modules or relying on modified kernels. This would also let the user defining the behavior of the program, making it specialized for the current applications and workloads, as opposed to fixed kernel implementations.

- **High-speed packet processing.** eBPF programs are JIT-compiled into native machine code before being executed[2], which provides more considerable advantages compared to an interpreted execution. Moreover, the execution

[2]This is valid only if the eBPF JIT flag is enabled in the kernel. However, latest kernel releases have this flag enabled by default.

of these programs at the XDP [75] level enables a high-speed packet processing and forwarding, with performance also close to user-level approaches, as shown in Figure 2.2a.

- **Excellent performance/efficiency trade-off.** As shown in Figure 2.2b, eBPF programs do not require polling to read network packets from the device hardware; they are triggered only when a packet is received. As a consequence, if the machine running the eBPF application is not receiving any traffic, no CPU cycles will be consumed as opposed to DPDK, where the CPU usage is always 100% even if no packets are received.

- **Integration with the Linux subsystems.** eBPF programs cooperate with the kernel TCP/IP stack, interact with other kernel-level data structures (e.g., FIB or neighbor table), and leverage kernel functionalities (e.g., TSO, skb metadata, etc.), possibly complementing existing networking features. Legacy applications or debugging tools can continue to be used without any change to the existing applications.

- **Security.** As opposed to custom kernel modules, eBPF programs cannot crash the system. The in-kernel *verifier* ensures that all the operations performed inside the eBPF programs are correct and safe, discarding the injection of faulty programs[3].

- **Explicit structure of NF operations.** Finally, eBPF programs have a clear distinction between stateless and stateful code; the operations outside the eBPF VM environment are carried out only through specific "helper" functions that have a well-defined syntax and construct. Multiple analysis frameworks developed over the years [166, 153, 126] have forced this assumption to simplify the analysis of NF operations to find buggy development semantic behavior (verifier safety does not imply "semantic safety") or to analyze the performance of a NF [85, 132]. eBPF programs have this assumption as part of the original design, making it easier to apply that type of analysis and concepts.

[3]Of course, a bug in the verifier [52, 51] can reverse this assumption but this can be considered as a remote scenario.

Chapter 3

Creating Network Service with eBPF: Experience and Lessons Learned

3.1 Introduction

The extended Berkeley Packet Filter (eBPF) is a recent technology that enables flexible data processing thanks to the capability to inject new code into the Linux kernel at run-time, which is fired each time a given event occurs, e.g., a packet is received. While its ancestor, the Berkeley Packet Filter (BPF) was used mainly to create packet filtering programs, eBPF has been successfully used primarily in monitoring tasks [69, 45, 28, 70]. Surprisingly, its usage in traditional network applications, such as data plane services, has been less intense.

In fact, the creation of complex network functions that go beyond simple proof-of-concept data plane applications has proven to be challenging, due to the several limitations of this technology, although it is evolving fast, as shown by the significant number of patches and new features added almost daily in the Linux kernel. In addition, eBPF is not (yet) backed by a rich ecosystem of tools and libraries aimed at simplifying the life of potential developers; the BPF Compiler Collection (`bcc`) [17] is more oriented to tracing than packet manipulation.

However, eBPF is very promising due to some characteristics that can hardly be found all together, such as the capability to execute code directly in the vanilla Linux kernel, hence without the necessity to install any additional kernel module; the possibility to compile and inject dynamically the code; the capability to support arbitrary service chains; the integration with the Linux eXpress Data Path (XDP) for early (and efficient) access to incoming network packets. At the same time, eBPF is known for some limitations such as limited program size, limited support for loops, and more, which may impair its capacity to create powerful networking programs.

This chapter presents our experience in developing complex network services with eBPF and shows the most promising characteristics of this technology as well as the main encountered limitations, as they appear in everyday life of a typical developer. This chapter will discuss the actual importance of the above limitations with respect to the necessity to create complex network applications and the possible solutions (if any). Finally, it will also discuss some of the peculiar advantages of this platform, backed by experimental evidence taken from our services. At the end, we ponder advantages and limits of the eBPF technology, analyzing its suitability as a platform for the development of future complex network services targeting mainly virtualized environments.

This chapter is structured as follows. Section 3.2 represents the central part of the chapter, highlighting our experience when coping with different aspects of eBPF, and the main lessons learned. Finally, Section 3.3 provides the necessary evidence to the previous findings and Section 3.4 concludes the chapter.

3.2 Experiences and Insights

This section presents the main challenges encountered while implementing complex network functions with eBPF, together with different insights we adopted (or could be adopted) to accomplish this task.

3.2.1 eBPF limitations

eBPF suffers from some well-known limitations due to its restricted virtual machine, which are needed to guarantee the integrity of the system. This Section discusses the impact of the above limitations and highlights some other, less known issues that arise when creating network services.

3.2.1.1 Limited program size

eBPF programs are executed within the kernel context; for this reason, their size is limited to a maximum of 4096 assembly instructions to guarantee that any program will terminate within a bounded amount of time. This restriction may be limiting when creating network functions that perform complex actions in the data plane, considering that the BPF assembly instructions generated after the code compilation could be significantly higher than the number of lines of code of the C source file.

Learning 1: This limitation can be circumvented by partitioning the network function into multiple eBPF programs and jumping from one to another through *tail calls*. This technique enables the creation of network services as a collection of

loosely coupled modules, each one implementing a different function (e.g., packet parsing, classification, fields modification), with a very low overhead in jumping from a piece to another, as shown in Section 3.3.5. This attractive feature also comes with an upper bound limit of 32 nested jumps, which we found being more than enough to implement complex services.

3.2.1.2 Unbounded loops

Since eBPF applications can be loaded at runtime in the kernel, they are checked through an in-kernel verifier that ensures programs cannot harm the system. The verifier looks for multiple possible threats (e.g., state of BPF registers, type of values, etc.), rejecting the code in case backward jumps are detected, thus ensuring that all programs will terminate. We list some cases where this limitation may be a problem.

- **Parsing nested headers**: this is the case of IPv6, which requires to loop through all the *extensions headers* to find the last header indicating the type of the upper-layer protocol in the payload of the packet. A similar issue affects MPLS and VLAN headers, whose number of instances is not known a priori. Creating a network function that performs these actions in eBPF is not possible unless we introduce additional constraints, as described below.
- **Arbitrary processing of packet payloads**: this is required for example to check the presence of a signature in the payload. While there are cases in which this loop can be avoided thanks to the availability of specific helpers that perform the entire job (e.g., to recalculate the L3/L4 checksum), in general the necessity to perform a loop scanning the entire packet cannot be excluded a priori.
- **Linear scanning of data structures**: algorithms that require a linear scan of data structures (e.g., a map) may need to be adapted for the eBPF environment. A possible example is a firewall that looks for the rule that matches a given packet, which is usually performed through a linear scan across all the active policies.

Learning 2: Although the eBPF assembly does not support backward jumps, as far as *bounded* loop are concerned, we can exploit the `pragma unroll` directive of the LLVM compiler to rewrite the loop as a repeated sequence of similar independent statements. This can be achieved by imposing a constant upper bound limit to the loop, such as the maximum number of IPv6 headers, nested MPLS labels,

packet size[1]. This solution presents two limitations: (*i*) the size of the program increases, with the possible consequences (and solutions) shown in 3.2.1.1; (*ii*) we may not be able to guarantee that all cases are handled, e.g., in case of an exceptional number of IPv6 headers is present. However, even if the lack of the support for unbounded loops seems to be an important limitation, we found it not so critical in our programs, as it can be often circumvented by creating bounded loops, although this is left to the responsibility (and experience) of the developer.

3.2.1.3 Send the same packet on multiple ports

This is still a rather common operation even in modern local area networks, e.g., to handle broadcast frames (e.g., ARP requests), multicast, or flooding (e.g., in an 802.1D bridge). However, at least three issues can be encountered when implementing this feature. First, we may need to loop through all the interfaces to forward the packet the desired number of times; this can be implemented only if we are able to set an upper limit on the loop and unroll it (as discussed in Section 3.2.1.2). Second, the packet must be cloned before sending it on an additional interface; this can be done with the `bpf_skb_clone_redirect()` helper, which simultaneously duplicates and *forwards* the original packet to a target interface. However, this helper is available only when the program is attached to the TC hook, while an equivalent helper is not available for XDP programs[2]. Third, if the service is part of a virtual chain composed by multiple NFs connected through tail calls such as in [1], the aforementioned approach fails. In fact, the redirect function will be followed by a tail call, which never returns the control to the caller, hence preventing the caller code to send a packet to multiple ports.

3.2.1.4 Packet-driven processing

While the execution of an eBPF program is triggered by an event, the only event that is supported by TC/XDP programs is a frame traversing the selected kernel hook. This prevents the eBPF data plane to react to other events such a timeout that signals the necessity to periodically send a packet (e.g., neighbour greetings in routing protocols), or to refresh an expired entry in a table. The above events have to be handled elsewhere, such as in the slow path (Learning 3) or in the control plane (Section 3.2.1.6).

---

[1] A patch [50] that adds support for bounded loops without the necessity to use the `pragma unroll` directive has been recently proposed and it may be integrated in future kernel versions.

[2] For the sake of precision, XDP offers only the `bpf_redirect_map()` helper, which sends the packet to a port but does not clone it.

3.2.1.5 Putting packets on hold

In some cases, network functions may need to put the current frame on hold while waiting for another event to happen. This is the case of a router that holds a packet while waiting for an answer to its own ARP request aiming at discovering the MAC address of the next hop; the original packet should be released after receiving the ARP reply. Unfortunately, eBPF does not have a "steal" action such as in Netfilter, hence preventing this technology to take the ownership of the packet. Possible workarounds to this problem can be envisioned, such as copying the entire packet in a temporary memory, but they may not be suited for all cases (e.g., handling retransmissions, as the packet has to be released when a timeout occurs).

Learning 3: Taken together, limitations 3.2.1.3-3.2.1.5 suggest the necessity to introduce a novel data plane component that is no longer limited by the eBPF virtual machine, which executes arbitrary code that can cope with the cases in which the current eBPF technology cannot be used. This brings to the evidence the necessity of a *slow path* module, executed in userspace, that receives packets from the eBPF program and reacts consequently with arbitrary processing defined by the developer; for example by modifying the packet and sending it back in the egress queue of a specific netdevice[3]. The necessity of this module is also highlighted in [157] where the authors use the OvS userspace module to process packets that do not match a flow in the OvS kernel eBPF data path.

3.2.1.6 No support for complex control planes

So far, eBPF has been used mostly for tracing applications, which feature a very simple control plane such as reading data from maps. As a consequence, existing eBPF software frameworks provide a nice set of abstractions that help developers to create data plane code (hook handling, maps, etc), while it is rather primitive with respect to the control plane, enabling userspace programs mainly to read/write data from maps. Networking services are rather different and often require a sophisticated control plane not just to read/write data from maps, but to create/handle special packets (e.g., routing protocols), to cope with special processing that may complicate (and slow down) the data plane if handled here (e.g., ARP handling; Section 3.2.1.5), or to react to special events such as timeouts (Section 3.2.1.4).

Learning 4: This results in non negligible difficulties when implementing the (complex) control plane of a service, as it forces developers to dedicate a considerable amount of time to write the code that handles common control plane operations from scratch, without any help from existing software frameworks.

[3]In Linux, a *netdevice* is a physical or virtual network interface card.

3.2.2 Enabling more aggressive service optimization

The traditional approach when implementing a network function is to *(i)* create a program that contains all possible use cases and control flows (branches) and *(ii)* make it completely agnostic with respect to its actual configuration, which is pushed in the data plane afterwards. With eBPF, this approach is no longer the only option; in fact, programs can be compiled from their C source code and injected in the kernel at runtime, with the system already up and running. This allows us to take advantage of the runtime service conditions (e.g., traffic pattern, service configuration, interface from which the traffic is received/sent) to empower more aggressive optimizations compared to traditional programs. This section presents three techniques of this type, enabled by the use of eBPF as data plane for our networking services.

3.2.2.1 Moving configuration data from memory to code

A conventional approach for configuring a network function is to save data (e.g., the public/private ports in a NAT, the set of rules in a firewall) in memory, which will be accessed by the run-time code each time a packet is received. In the eBPF domain, this corresponds to saving data in *maps*, which provide a bidirectional userspace-kernel communication channel. While this approach is the only viable option for other technologies, eBPF enables the loading of new code dynamically, hence allowing the creation of situational-specific code that also embeds the data needed for the current processing. This technique leverages the superior processing capabilities of modern CPUs (e.g., speculative execution), trading more processing instructions for fewer memory accesses, which are known to introduce a noticeable penalty in particular when random data access patterns are required, which lead to cache ineffectiveness.

Learning 5: Hardcoding parameters in the eBPF code in a way that the service can directly use them without any explicit memory access may lead to significant performance gains (Section 3.3.3.2). However, this requires to handle configuration changes by *dynamically reloading* the program with the updated parameters; more details will be presented in Section 3.2.2.3.

3.2.2.2 Code tailoring

A network function can have a different set of features that are not always needed at runtime. For example, our bridge supports both VLANs and Spanning Tree, but they may not be required (hence be turned off) at a given time. The amount of code needed to handle these features is not negligible and can impact the forwarding performance of the eBPF network function.

Learning 6: Our experiments showed that cutting the superfluous code, at runtime, will bring a significant reduction in the number of control flows and branches of the program, hence simplifying the new code and improving the overall performance of the service, as shown in section 3.3.3.1.

3.2.2.3 Dynamic reloading

The previous two techniques can provide substantial performance gains; however, their value would be impacted without the possibility to *reload* the program *at runtime* with a more appropriate version, while maintaining at the same time the state (e.g., maps) and configuration of the old program. Dynamic code reloading is currently supported in eBPF, but the existing software frameworks do not offer any help, leaving this responsibility in the developer's hands and hence requiring additional complexity when writing efficient network services. Our prototypical code that supports this feature is strongly hinged on reducing the service disruption and packet loss (see Section 3.3.3). While the new service is compiled and injected, the old one still handles the traffic. When the new program is ready, maps of the old instance are attached to it and then atomically *swapped* by substituting the pointer to the old program with the new one. At this point, the new program will start processing the traffic, and the old one is unloaded.

3.2.3 Data structures

eBPF does not have the concept of "raw" memory as used by classical computers; data are in fact stored in memory areas structured according to a predefined access model (e.g., hash map, lru map, array). As of this writing, there are seventeen types of map that can be used by an eBPF program. Even though the existing set of maps is very large and allows to fulfill the requirements of the majority of applications, in the next two subsections we present some cases in which it may not be enough.

3.2.3.1 Stack map

We may envision a service that needs to maintain a pool of elements that can be consumed (e.g., through a *pop* action to get the first free element of the pool) or produced (e.g., a *push* operation to insert an item back in the pool) atomically, hence similar to the behavior of a *stack*; this is the case of a NAT service, which needs to keep the list of available TCP/UDP ports. Unfortunately, this type of data structure is not present among the set of maps available in eBPF. Although its behavior can be emulated using an array and a global counter, used as the index of the first element to retrieve, it is subject to concurrency problems when multiple instances of the same program access the same data from different kernel threads, causing race conditions.

3.2.3.2 Map with timeout

A typical scenario for networking functions is to have entries in a table with an associated timeout; when an entry is not accessed for a specific time interval, it expires and is removed from the list (e.g., the filtering database of a bridge). Unfortunately, eBPF does not have such a map. This behavior can be emulated by *(i)* inserting an additional field in the entry that corresponds to the current timestamp and *(ii)* check, at every access, that the item has expired; if so, the entry is deleted and the service continues as if the entry was not present. Obviously, entries that are no longer accessed will never be deleted unless an LRU (least recently used) map is used. This approach partly complies the lack of this table in eBPF (indeed, it is the approach used to implement our 802.1D bridge), even if it complicates the data and control plane of the service that must take care of discarding old entries, with possible racing conditions.

3.2.3.3 Concurrent map access

When a hook triggers an eBPF program in the kernel, multiple instances of the same eBPF application can be executed simultaneously on different cores. A normal eBPF map has a single instance across all cores and could be accessed simultaneously by the same eBPF program running on different cores. eBPF maps are native kernel objects that are protected through the kernel Read-Copy-Update (RCU) [163] mechanism, which makes their access thread safe, regardless of whether the interaction occurred from userspace or directly from the eBPF program. The fact that map access is thread-safe does not exclude the presence of data races, given the implicit multi-threading capabilities of eBPF and the impossibility to use locks.

The interaction between the control plane and the data plane is also subject to race conditions since they do not have a standard synchronization mechanism. For example in a network bridge, if the cleanup of the filtering database is performed in the control plane, we could create the following situation: (*i*) the control plane reads an entry from the filtering database and realizes that it is too old, so it must be removed, (*ii*) the data plane receives a packet for that entry and updates the filtering database with the new timestamp, (*iii*) the control plane eliminates the newly inserted entry, producing an unexpected behavior.

Learning 7: We have noticed that map access is thread-safe since these structures are protected by the RCU mechanism. However, a race condition can still happen either between control and data plane or from the same eBPF program running on different cores. Unfortunately, we have not yet found a definitive and general solution for all cases. It is, therefore, the developer who has to take care of this problem and find alternative solutions depending on the application logic.

3.2.4 High performance processing with XDP

XDP provides a mechanism to run eBPF programs at the lowest level of the Linux networking stack, directly upon receipt of a packet and immediately out of driver receive queues. It has two operating modes; the first one, called *Driver (or Native) mode*, is the primary mode of operation; to load eBPF programs at this level, the driver of the netdevice must support this model. Running network applications in XDP produces significant performance benefits (as shown in Section 3.3.4) since the application can perform operations on the packet (e.g., redirect, drop or modify) before any allocation of kernel meta-data structures such as the `skb`, spending fewer CPU cycles for processing the packet compared to the conventional stack delivery. The second one, called *Generic (or SKB) mode* allows using XDP within drivers that do not have native support for it, providing a simple way to use and test XDP programs with less dependencies.

In section 3.2.4.1 we show the main differences between XDP and the other network hook point, i.e., TC; we will then present the main drawbacks found in the current support of the Linux kernel for both XDP Driver (section 3.2.4.2) and XDP Generic mode (section 3.2.4.3).

3.2.4.1 Limited helpers

XDP programs are only allowed to call a subset of helpers compared to eBPF services attached to the Traffic Control (TC) layer. In general, eBPF has a set of base helpers (e.g., map lookup/update, tail calls) available for all types of programs, with some specific helpers for each category of hook; approximately 29 available in TC and only 7 in XDP. We summarize the main differences in the following list:

- *Checksum calculation:* Primitives for checksum computations were not fully available in XDP as they are in TC. It is going to be fixed in the upcoming version 4.16 [31] of the Linux kernel, but this prevents an eBPF program exploiting this feature to be executed on an older kernel. In that case, the solution is to recompute the checksum "by hand", with dedicated code in the XDP program.

- *Push/Pop headers:* XDP does not offer any helper to push and pop a VLAN tag from the packet or to perform tunnel encapsulation or decapsulation. In case this feature is needed, the XDP program can use the more generic `bpf_xdp_adjust_head()` helper, which provides the ability to adjust the starting offset of the packet along with its size, so it is possible to manipulate the packet according to the application logic.

- *Multi-port transmission:* As already highlighted in 3.2.1.3, an equivalent of the `bpf_skb_clone_redirect()` helper available in TC is missing in XDP. This does not allow to forward a packet on several ports at the same time,

which is required by different network applications (e.g., bridge, router), unless we implement this feature in the *slow path*.

Learning 8: Writing programs with XDP does not have significant differences compared to TC; most of the actions such as direct packet modification, access to maps or the use of tail calls remain identical with the other hook points. However, the limited number of helpers forces the developer to use more generic functions or to implement those functionalities in the slow path, complicating the code and making it less portable, with the consequence that the code must different depending on the kernel hook to which it is attached.

3.2.4.2 XDP Driver mode limitations

Most XDP-enabled drivers today use a specific memory model (e.g., one packet per page) to support XDP on their devices. Among the different actions allowed in XDP, there is the possibility to redirect the packet to another physical interface (`XDP_REDIRECT`). While this action is currently possible within the same driver, in our understanding, it is not possible between interfaces of different drivers. The main problem is the lack of a common layer/API that the drivers can use to allocate and free pages. With this model, when a driver performs a packet transmission, it can communicate the actual sending, back to the Rx driver, which may recycle the page without having to run into costly DMA unmap operations[4]. This lack of generality limits the network applications that can take advantage of the speed gain provided by XDP, that have to entrust the normal stack processing to make the correct forwarding of the packet.

3.2.4.3 Generic XDP limitations

As previously mentioned, XDP generic can be used to run XDP programs even on drivers that do not have native XDP support. Although they are executed right after the `skb` allocation, thus losing the advantages available in the driver mode, it still provides better performance than other hook points such as TC, as shown in section 3.3.4. When triggered, XDP generic programs can modify the content of the received packet; however, if the packet data are part of a *cloned skb*, an XDP program cannot be executed on this packet, since cloned skb cannot be modified. This leads to some limitations such as handling TCP traffic; in fact, our network function running on the XDP generic hook was never be able to receive TCP traffic, since most of the packets belonging to TCP sessions are cloned, in order to be later retransmitted, if necessary.

[4]An interesting discussion on this topic is available here [37]. A patch towards that direction is available in [38].

Learning 9: Although writing programs compatible with both XDP and TC is not a significant problem, their use is not interchangeable. Using XDP programs as substitutes of TC services is not always possible, resulting advantageous only for specific applications. For example, connecting containers with XDP services may not be appropriate since most of its advantages given by the the early stage in which frames are captured would be lost. The XDP hook has been designed to work mainly in ingress, making tricky the modelization of services such as a firewall that would need, for example, to capture packets generated by the host and going outside the network interface; this is instead possible using TC as a hook point in ingress and in egress.

3.2.5 Service function chaining

The possibility to connect eBPF programs through *tail calls* in kernel facilitates the combination of network services (e.g., bridge, router, NAT) in a virtual chain, with considerable advantages as shown in [1]. In this way, eBPF services can connect either *(i)* through a netdevice or *(ii)* through a tail call, directly to another eBPF module. However, this requires the creation of a different source code based on the port the eBPF program is attached to, since the assembly instructions used in the two cases are different, which is an unnecessary complication for a developer.

Learning 10: To make the internal logic of the service independent from the connection type, we can introduce the concept of *virtual port*, which is used by the network function to receive and forward the traffic, and we can dynamically generate the proper source code for any given port. However, the creation of this level of abstraction, so that eBPF programs are independent from the type of interconnection with the outside world, is certainly possible but requires a significant effort of the programmer as it is not explicitly foreseen by any available framework.

3.3 Experimental Evaluation

This Section provides experimental evidence about the topics discussed in Section 3.2, showing the impact of the main eBPF limitations and the improvements made in our prototypes. This evaluation leverages some of the network services we have implemented, hence exploiting real applications as a test-bench for our measurements.

3.3.1 Test environment and evaluation metrics

Our testbed encompasses two machines (Intel i7-4770 CPU running at 3.40GHz, four cores plus hyper-threading, 8MB of L3 cache and 32GB RAM) physically

connected to each other through two direct 10Gbps links terminated in an Intel X540-AT2 Ethernet NIC. Both machines feature an Ubuntu Server 16.04.4 LTS, kernel 4.14.1[5], with the eBPF JIT flag enabled.

Throughput was tested by generating a unidirectional stream of 64B packets through Pktgen-DPDK 3.4.9, with a rate that is dynamically adjusted to achieve no more than 1% packet loss; depending on the test, packets may be looped to the sender machine. Latency tests were carried out with Moongen [58], which generates the same traffic pattern as before but it exploits the hardware timestamp on the NIC to determine the traveling time of a frame when returns back to the sender; by default, one frame every millisecond is sampled. All tests were repeated ten times and the figures contain error bars representing the standard error calculated from the different runs.

Unless explicitly stated, all tests generate traffic so that only one CPU core is involved in the processing. Consequently, throughput in case of real deployment can be much higher than the reported values thanks to the session-based traffic load balancing provided by the Linux kernel, which automatically exploits multiple CPU cores for the processing.

3.3.2 Overcoming eBPF limitations

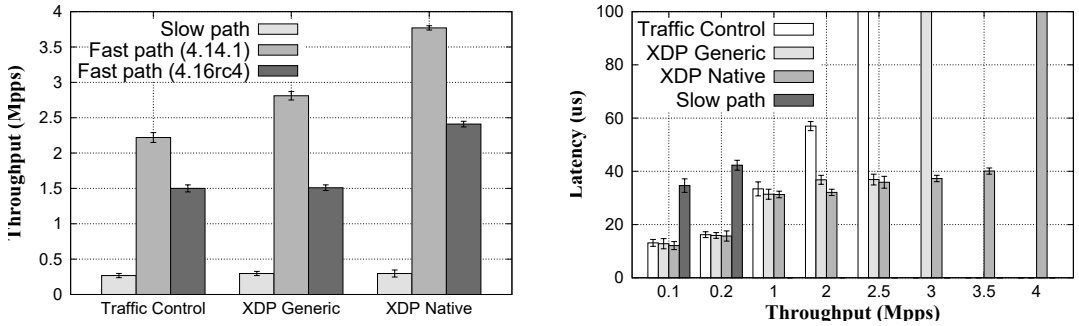
3.3.2.1 Slow-path forwarding performance

Section 3.2.1 showed the most important eBPF limitations that prevent the implementation of all the features required by complex network applications in the data path, which can be delegated to a more flexible userspace component, although with reduced performance.

To validate this module, we used our 802.1D bridge with two ports connected to the physical interfaces of the machine under test. The bridge service uses the slow path when a packet for an unknown MAC destination is received; in that case, it will be sent to the slow path module, which floods that packet to all output ports. To test this feature, we forced our bridge to send each packet to the slow path, from where they are forwarded to the output port.

Figure 3.1a shows the throughput calculated in different networking hook points, i.e., Traffic Control, XDP Generic and XDP Native between the slow path and the fast path. We notice that, while the fast path forwarding varies depending on the hook point used (XDP performs better than the Traffic Control), the same is not valid for the slow path; the latter, in fact, uses the same mechanism to send the packet to userland regardless the hook point type to which the eBPF program is

[5]We noticed that with newer kernels (e.g., 4.15/6) there is a marked performance deterioration, as shown in Figure 3.1a, supposedly due to the fixes introduced after the Meltdown and Spectre vulnerability disclosure.



(a) (b)

Figure 3.1: Throughput (left) and latency (right) for the Bridge service when redirecting packets entirely in the fast path and when using the slow path.

attached, hence representing the bottleneck for this test. Moreover, we notice that processing a packet in the slow path takes about three times more time than the same processing done in the kernel. In this case, the maximum forwarding rate is about 0.3Mpps regardless of packet size, since most of the time is spent in moving the packet to userspace (and back), significantly reducing the forwarding speed.

Figure 3.1b indicates the latency measurements calculated in the same conditions as before; we can notice that as long as the traffic remains below the maximum throughput calculated for each hook point, the latency values are almost the same regardless the hook type; this is due the batching mechanisms adopted by the driver that introduce a fixed cost on the packet processing. On the other hand, slow path processing incurs in a higher overhead due to the additional copy of the packet between kernel-userspace. Also, we notice how the latency grows when approaching the maximum value calculated in the previous test; in this case, the packet loss value increases considerably with a consequent delay in the packet processing. For example, the throughput of the bridge service calculated in Traffic Control is 2.22Mpps, and the peak in latency is seen at 2.5Mpps (in the graph, we do not show the absolute value for readability reasons, but it is, in all three cases, over one millisecond). The same behavior is identified in XDP Generic and XDP Native, where the spike is at 3Mpps and 4Mpps respectively. These numbers indicate the importance of performing most of the actions in the eBPF fast-path, reducing the number of trips to the slow path. It is worth noticing that in real cases we have seen few packets to be raised as exceptions and sent through the slow path, being closer to 100% of the packets handled in the kernel for normal applications.

3.3.3 Enabling more aggressive service optimization

3.3.3.1 Code tailoring

To evaluate the potential benefits of this technique we took our bridge service and we changed its features at runtime, enabling the VLAN and Spanning Tree (STP) and measuring the final throughput using the same traffic, independently from the enabled feature. Figure 3.2a shows how the complexity increases, in terms of BPF instructions, when functionalities requiring more complex actions are enabled. For instance, handling STP requires to parse and recognize BPDUs, requiring also the intervention of the control plane. This complexity is then reflected in the overall forwarding performance of the service, where we can see a drop of up to 20% of the throughput between the baseline, in which acts as a simple L2 learning Bridge, to the full version of the bridge that supports VLAN and STP. Note that this performance improvement would not be possible without the *code tailoring* and the *dynamic reloading*, forcing the user to work with the full version of the bridge even if those features were not required at that moment.

3.3.3.2 Moving configuration data from memory to code

To estimate the goodness of this technique, we used our bridge service, comparing the actual throughput in the three different hook points, when the optimization is enabled and when it is not, as shown in Figure 3.2b. In this specific case, the optimization consists in moving the content of the filtering database from memory (i.e., a map) directly into the code, with the entries statically embedded in it, for example via a switch-case on the destination MAC address to forward the packet on the right port. Obviously, we set a maximum limit to the entries directly written in the code. This feature permits to optimize the most common case, by statically inserting the entry within the code, hence avoiding (costly) memory accesses, with performance benefits ranging from 5% or 10%, with the possibility of obtaining higher gains by combining this technique with the previous one.

3.3.3.3 Reloading

This technique is heavily used in our services (in conjunction with the ones described above) to adjust the code injected in the kernel with the updated runtime service parameters. Table 3.1 shows the cost of this technique by comparing the reloading time for different services, which can be split into two pieces. The first one is the time needed to compile the C source code into eBPF assembly instructions while the second is the time required to inject the program in the kernel, which involves a pass in the in-kernel verifier.

We notice how the compilation phase consumes most of the reloading time (we leverage `bcc` [17] to appropriately package eBPF modules in the kernel) while only

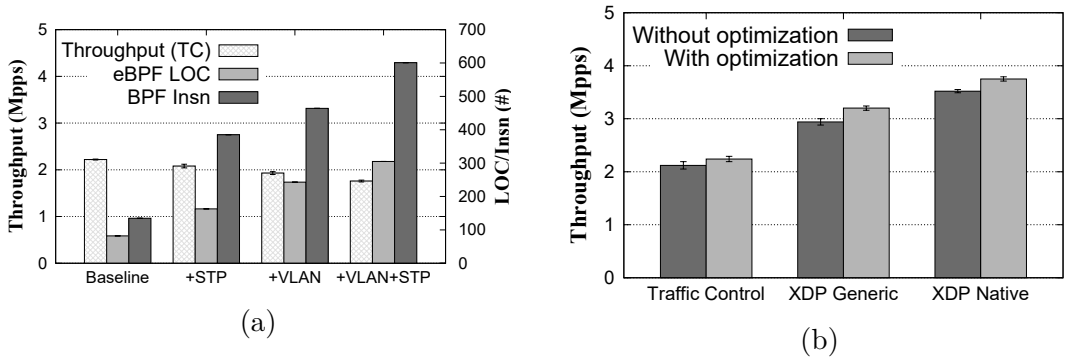


Figure 3.2: Effect on the end-to-end throughput, using the code tailoring technique (left) and the moving configuration from memory to code (right).

Table 3.1: Reloading time of various eBPF services

Service	C LOC	BPF Insn	Compilation (ms)		Injection (ms)
			Kernel .h	UAPI .h	
Firewall*	1094	1564	8683	850	50
LBDSR	305	910	889	830	6
LBRP	470	723	885	853	2
Router	331	458	885	799	1
Bridge	243	464	854	236	2
NAT	564	441	847	809	1
DDoS	136	74	806	642	1

* This service uses a chain of eBPF programs; the time shown refers to the one needed to compile and inject the entire programs chain, which comprises several eBPF programs.

a small part, around 1%, is spent on the in-kernel injection. Also, we realized that using the Linux standard kernel headers introduces a considerable overhead as they internally include other headers that often are not needed by our eBPF program. By carefully selecting the headers present in the `uapi` directory, which are often smaller and more compact, we noticed a significant reduction in compilation time, as shown in the firewall service in Table 3.1. In fact, this service uses multiple eBPF programs in the data path; optimizing the compilation time of every small program reduces the overall reloading time of the service. It is important to notice that `bcc` provides additional primitives to facilitate the interaction with the eBPF ecosystem; during the compilation phase, the code is then rewritten mapping the `bcc`-provided

helpers into the corresponding eBPF functions. The compilation time shown in the Table 3.1 is the sum of three phases (of approximately the same duration) during which the code is pre-processed, rewritten and ultimately compiled producing the final BPF assembler code.

Some works such as [42] keep compiled versions of their services and then perform an optimization directly on the compiled code, without this additional overhead. However, we believe that this mechanism limits the potential of previous techniques, reducing their possible optimizations. Although in this case the overall reloading time would not be negligible, as we explained in Section 3.2.2.3, we swap the existing program with its optimized version after the program has been compiled and injected, thus avoiding any service disruption.

3.3.4 High performance processing with XDP

XDP Native. The performance benefits of Native XDP services are evident from the previous figures. In fact, attaching the same program in XDP Native mode leads to an increase in performance of about 65% (Figures 3.1a-3.1b), allowing to achieve higher throughput. In addition, when comparing XDP programs with the other hook points at the same throughput, it brings to a significant reduction of CPU consumption due to the lower overhead for packets management. This speed comes with the limitation that programs of this type can only be used when the entry points of the chain are physical or virtual interfaces[6], while they must return to the normal stack delivery in case of different workloads (e.g., containers).

XDP Generic. Previous figures show that XDP Generic provides about a 25% of performance improvement compared to Traffic Control (TC), allowing us to conclude that it can be used to speed up the performance of services even for drivers that do not have native XDP support. Generic XDP programs may be directly attached to virtual ethernet interfaces (veth) providing services to workloads such as containers, with hopefully a performance increment. However, this feature may result difficult to apply in real-world scenarios, due to the problem mentioned in Section 3.2.4.3.

3.3.5 Service function chaining

The ability to directly connect eBPF services to each other in the kernel (section 3.2.5) is a significant advantage of this technology. This section evaluates the overhead of the tail calls in the three different hook points by using a simple program that forwards a packet between two physical interfaces, but whose code

[6]Recently, support for receiving frames with XDP has been added to the tuntap driver [162].

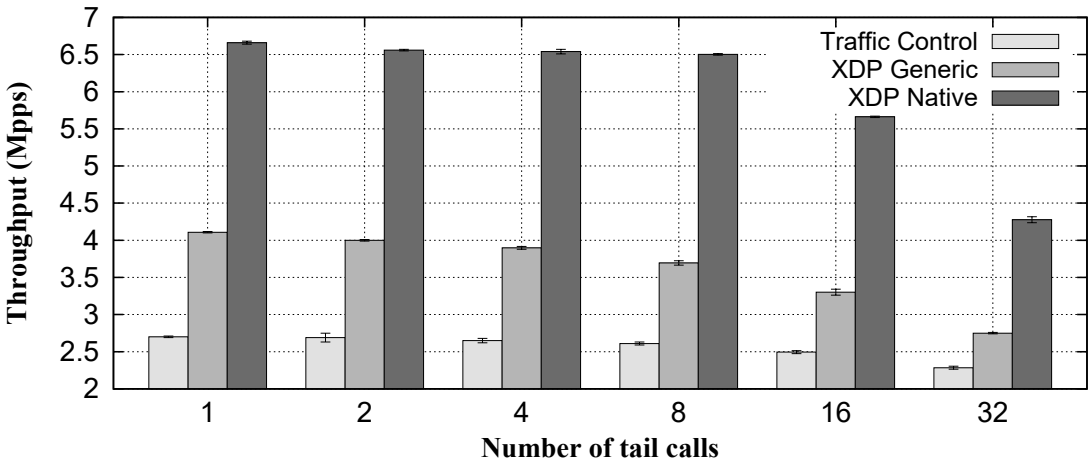


Figure 3.3: End-to-end throughput with an increasing number of tail calls.

includes a growing number of (empty) tail calls. As shown in Figure 3.3, the overhead of the tail calls is almost negligible up to 8, which is enough for most of the applications, while it increases significantly with 16 and 32; the reason of this additional overhead is given by the presence of indirect jumps, which are converted to *retpoline* [30] to protect against the branch target injection vulnerability disclosure (a.k.a., Spectre).

3.4 Conclusions

This chapter presents our experience in developing complex network applications with eBPF, an up-and-coming technology that allows executing code at runtime in the Linux kernel, without the need to package custom kernel modules. We described the main limitations of this technology demonstrating how, in most cases, these can be circumvented without affecting the necessities of real network applications. In other cases, we have proposed alternative solutions to overcome these limitations. Thus, we identified and discussed several ideas related to the possibility of injecting code dynamically in the kernel that opens the way to several new optimizations strategies. Finally, we verified the real applicability of the proposed ideas and the consequent performance advantages, exploiting the several applications we created for our experiments. We are confident that this type of practical learnings can positively influence ongoing development and advancements in the field of eBPF-based network services and applications.
Note: The eBPF subsystem is in continuous development and new features, as well as performance improvements, are added daily. As a consequence, it is possible that some performance results or limitation presented in this chapter are now obsolete. For instance, support for stack and queue maps was added recently, the overhead

given by consecutive tail calls has been almost removed [32] and the maximum number of instructions for every eBPF program has been increased to 1 million [150]. On the other hand, most of the limitations and results showed in this work are still valid since they are part of the security model of the eBPF subsystem.

Chapter 4

Polycube: A Framework for Flexible and Efficient In-Kernel Network Services

4.1 Introduction

Network Functions Virtualization (NFV) enables network services to be transformed in pure software images that are executed on standard servers. This technology guarantees lower costs thanks to the reduction of the number of physical appliances [59] and to the possibility to rely on (cheap) commodity hardware. At the same time, it enables more agile services thanks to the click-and-play nature of the software.

The most common approach to NFV is through a set of (chained) VMs or containers, connected by means of a virtual switch [127]. This often includes heterogeneous applications, built from different vendors, with diverse characteristics e.g., in terms of configuration protocols and life-cycle management, which complicates day-by-day operations [123]. This heterogeneity impairs also on the possibility to achieve higher throughput through cross-NF optimizations, as each application operates in isolation. For instance, even simple approaches, such as zero-copy or shared memory between cascading functions (e.g., [20, 79, 123, 134]), are often very difficult to deploy in practice. Furthermore, advanced features such as service decomposition [96], fault-tolerance [143], high availability [67, 129], should be provided separately for each VNF, complicating the design of the control plane and making the system more complex [63]. Finally, the computational requirement for the above VNFs is often huge, due to the number of different components involved (e.g., hypervisors, VMs with their guest operating systems, vSwitch, etc.) not to mention the cost in moving a packet during its journey, due to the many components traversed and the several transitions between kernel and user space.

Driven by the above-mentioned issues, several NFV frameworks [125, 73, 123]

have been presented with the goal of providing both a programming model (for building NFs) and an execution environment (for running NFs) together with a set of common abstractions that avoid re-implementing same functionaly across different services. The majority of the state-of-the-art NFV frameworks rely on kernel-bypass approaches (e.g., DPDK [54], Netmap [133]) to provide high-level performance in terms of throughput and latency.

Although they bring unquestionable performance improvements, there are some scenarios where they may not be appropriate or their performance result limited, as we have seen in the previous chapters. On the other hand, the alternative to them is to perform the processing entirely in the kernel. A few years ago, this would have been a "dumb" idea; the monolithic design of the Linux kernel was a great limitation for two desired properties of network function: *performance* and *flexibility*. Network applications relying on kernel components (e.g., OvS [127]) have been proved hard to evolve due to the cost of managing kernel modules or to deal with a large and complex codebase, "convince" maintainers of its usefulness, and wait for a new release; a process that may take several months, or years. However, the recent introduction of the extended Berkeley Packet Filter has introduced a new type of application deployment method for the Linux kernel, resuming the opportunity to achieve *(i)* the desired flexibility, thanks to the possibility to dynamically inject userspace defined programs into the kernel without having to restart the machine or rely on custom kernel modules, and *(ii)* performance, since it is possible to "skip" specific sections of the TCP/IP stack, if not needed.

In this chapter we present Polycube, a novel software framework that enables the creation, deployment and management of **in-kernel** network functions. In particular, it offers *(i)* the possibility to implement complex functions where the data plane is executed in the kernel context, *(ii)* enables the creation of arbitrary service chains, hence simplifying the creation of complex services through the composition of many elementary components and *(iii)* offers a service-agnostic interface that decouples the control and management logic (which is generic and valid for all services) from the actual service logic, hence enabling the dynamic and seamless deployment of arbitrary network services. In summary, this chapter makes the following contributions:

- We show the design and architecture of Polycube (section 4.3). Although Polycube contains several ideas and concept that are shared among the existing NFV frameworks (e.g., modularity, functional service decomposition), it applies them to the *kernel* world, which implies different challenges and design solution to be achieved in this scenario. To the best of our knowledge, Polycube is currently the only opensource framework that provides a unified architecture for the development, execution and chaining of *in-kernel* network functions.

- We provide a description of the APIs and abstractions provided to the developers to simplify the development of new services (section 4.4).
- We demonstrate the practical benefits and of the Polycube programming model with a complex application, namely a network provider plugin for Kubernetes (section 4.8).

Polycube source code, documentation and implementations of the various network services are available at GitHub [12].

4.2 Design Goals and Challenges

The main objective of Polycube is to provide a common framework to network function developers to bring the power and innovation promised by NFV to the world of in-kernel packet processing. This has been made possibile in the recent years thanks to the introduction of eBPF. However, eBPF was not created with this goal in mind; it serves only as a generic virtual machine that enables the execution of user-defined program into the kernel, attaching them to specific points into the Linux TCP/IP stack or to generic kernel functions (e.g., kprobes).

Common structure and abstractions for developing in-kernel NFs. The eBPF subsystem and, more generally, in-kernel applications do not have the concept of virtual ports from which the traffic is received or sent out. Moreover, as we have seen in Chapter 3, the realization of *complex* network functionalities are not always possible in eBPF, given its security model that is forced by the in-kernel verifier to ensure that the execution of the program does not harm the system. For instance, no abstractions current exist to implement the (complex) control plane of a service, hence forcing developers to dedicate a considerable amount of time to handle common control plane operations (e.g., user-kernel interaction).

Polycube must provide a common programming framework and models to allow developers to use high-level abstractions to solve common problems or known limitations in a efficient and transparent way, while the framework optimizes the implementations of those abstractions, ensuring high performance.

Simple management and execution of the NFs. Polycube must allow external operators (e.g., SDN controllers, orchestrators, network administrators) to configure in-kernel functionalities to support a diverse set of use cases. This implies the possibility to compose and configure the datapath functionalities or to dynamically upgrade or substitute a given service at runtime. A clear separation between the in-kernel data plane and the control plane would be desired, so that is becomes easier to dynamically regenerate and reconfigure the data path to implement the user policies.

Programmable and extensible service chaining. In a NFV environment, a packet is typically processed by a sequence of NFs, giving the possibility to run different NFs at the same time, which are also manufactured by multiple vendors. Polycube must enable the possibility to create chain of NFs in the kernel, guaranteeing the correct forwarding sequence and same degree of isolation between them.

4.3 Architecture Overview

This Section introduces first the main ideas that inspired the design of Polycube; then it will present the resulting software architecture and the most significant implementation details.

4.3.1 Unified Point of Control

All network functions within Polycube feature a unified point of control, which enables the configuration of high-level directives such as the desired service topology. In addition, it facilitates the provisioning of cross-network function optimizations that could not be applied with separately managed services. Polycube supports this model through a single, service-agnostic, userspace daemon, called `polycubed`, which is in charge of interacting with the different network function instances. Each different type of virtual function is called `Cube`, which are similar to *plugins* that can be installed and launched at runtime. A new Cube can be easily added to the framework within a specific registration phase, in which the service sends the information required for its identification, such as the service type or the minimum kernel version required to run the service. When the service is registered, different instances of it can be created by contacting `polycubed`, which acts mainly as a proxy; it receives a request from a northbound `REST` interface and forwards it to the proper service instance, returning back the answer to the user.

4.3.2 Structure of Polycube services

Each Polycube service is made up of a *control plane* and a *data plane*. The *data plane* is responsible for per-packet processing and forwarding, while the *control and management plane* is in charge of service configuration and non-dataplane tasks (e.g., routing protocols). Although this separation between the control and data plane is common in many network functions architectures, Polycube provides a clear separation between these components; each service is composed of a set of standard parts that make it easier for the programmers to implement the desired behavior, while Polycube takes care of creating all the surrounding glue logic, handling all the interactions and communications between the different components.

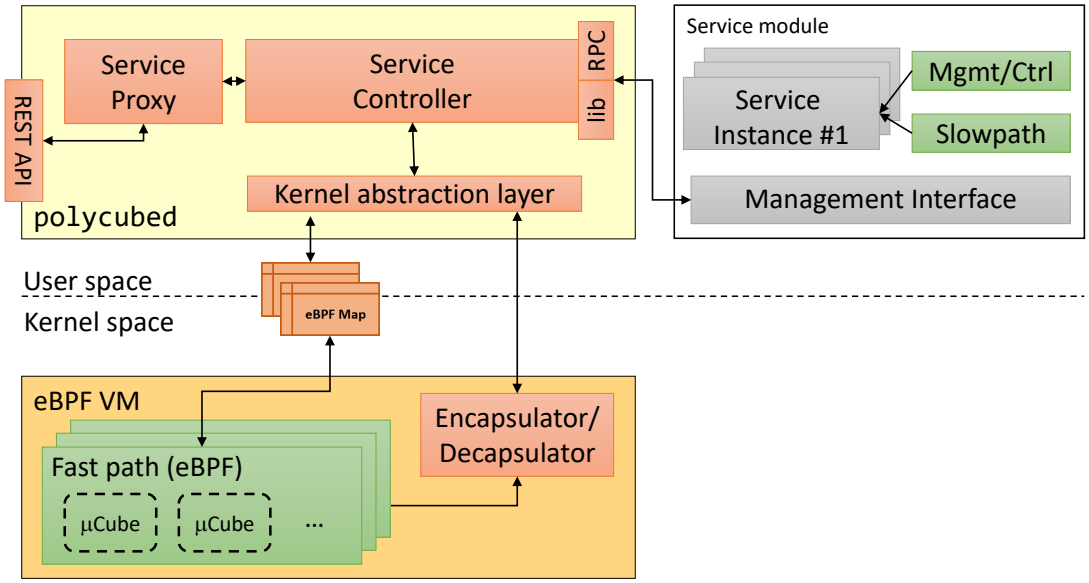


Figure 4.1: High-level architecture of the system

4.3.2.1 Data plane

The data plane portion of a network service is executed *per packet*, with the consequent necessity to keep its cost as small as possible. A Polycube service data plane is characterized of a *fast path*, namely the eBPF code that is injected into the kernel, and a *slow path*, which handles packets that cannot be fully processed in the kernel or that would require additional operations, slowing down the processing of the other packets.

Fast path. When fired, the fast path retrieves the packet and its associated meta-data from the receive queues, then it executes the injected eBPF instructions. Typical operations are usually very fast, such as packet parsing, lookups in memory (e.g., to classify the packet), and map updates, such as storing data in memory (e.g., statistics), for further processing. When those operations are carried out, the fast path returns a forwarding decision for that particular packet or send it to the slow path for further processing.

Polycube supports the composition of a service fast path as a set of micro-functional blocks, which can be created by stitching multiple eBPF programs together, controlled and injected separately from the control plane using the Polycube service-independent APIs. This set of eBPF micro-blocks, called *micro-cubes (µCubes)*, are part of a single Cube instance and are handled by an unique control plane and slow path module, as opposed to different Cube instances that have their own controllers. This modular design introduces the necessity to specify an order of

execution of the μCubes inside the NF; when a packet reaches a Cube composed of different micro-blocks, Polycube has to know the first module to execute, which in turn will trigger the execution of the others μCubes within an arbitrary order based on its internal logic. To do this, Polycube introduces the concept of `HEAD` μCube, which is unique within the Cube itself and represents the entry point of the entire service, whose execution is triggered upon the reception of a packet in a port. Then, it can "jump" to the others μCubes. Having a different set of eBPF programs, each one performing a specific function, is useful in particular for two reasons. First, it allows the developer to handle each feature separately, enabling the creation of loosely coupled services with different functionalities (e.g., packet parsing, classification, field modification) to be dynamically composed and replaced; each single μCube can be substituted at runtime with a different version or can be directly removed from the chain if its features are not needed anymore. Second, it can be useful to overcome some well-know eBPF limitations such as the maximum size of an eBPF program or the inability to create unbounded loops in the code.

Slow path. Although eBPF offers the possibility to perform some complex and arbitrary actions on packets, it suffers from some well-known limitations due to his restricted virtual machine, which however are necessary to guarantee the integrity of the system. Those limitations may impair the flexibility of the network function, which *(i)* may not be able to perform complex actions directly in the eBPF fast path or *(ii)* could slow down its execution, adding more instructions in the fast path to handle exceptional cases. To overcome those limitations, Polycube introduces an additional data plane component that is no longer limited by the eBPF virtual machine and it can hence execute arbitrary code. The *slow path* module is executed in userspace and interacts with the eBPF fast path using a set of components provided by the framework. The eBPF fast path program can redirect packets (with custom meta-data) to the slow path, similar to Packet-In messages in OpenFlow. Similarly, the slow path can send packets back to the fast path; in this case, Polycube provides the possibility to inject the packet into the *ingress* queue of the network function port, simulating the reception of a new packet from the network, or into the *egress* queue, hence pushing the packet out of the network function.

4.3.2.2 Control and management plane

The control plane of a virtual network function is the place where out-of-band tasks, needed to control the data plane and to react to possible complex events (e.g., Routing Protocols, Spanning Tree), are implemented. It is the point of entry for external players (e.g. service orchestrator, user CLI) that need to access service's resources, modify (e.g., for configuration) or read service parameters (e.g., reading statistics) and receive notifications from the service fast path or slow path. Polycube

defines a specific *control and management* module that performs the previously described functions. It exposes a set of `REST` APIs used to perform the typical CRUD (create-read-update-delete) operations on the service itself; these APIs are automatically generated by the framework starting from the service description (i.e., a YANG model of the service), removing this additional implementation overhead to the programmer. To interact with the service, an external player has to contact `polycubed`, which uses the service instance, contained in the URL, to identify which service the request is directed and dispatches it to the corresponding service control path, which in turn serves the request modifying its internal state or reflecting the changes to the service data path instance. More details on how the control plane and the REST APIs of each service are automatically generated is presented in Section 4.6.

4.3.3 Remote vs Local services

The separation between the data and control plane allows to execute the two components separately, not necessarily on the same server. While the former is running in the server, the latter can be executed either *locally* or *remotely*. Polycube may support both *local* services, installed as local applications on the same server of `polycubed` and whose interaction is through direct calls, and *remote* services, possibly running on a different machine and communicating with `polycubed` through RPC mechanism. When a new service is plugged into the framework, it communicates to `polycubed` if it is local or remote. In the first case, the path to the service executable (i.e., a dynamic library file) is specified and `polycubed` loads the library at runtime, forwarding the requests to the service control path as a normal function call. In the second case the service is registered by providing the remote RPC endpoint; in that case all subsequent requests for that service will be redirected through the RPC channel. Polycube provides a *management interface* that allows to control any service data plane, regardless of the service type and structure, hence being agnostic to the control plane location. It allows to get access to any registered service in the same way from its `REST` interface, facilitating the service developer who does not have to deal with the low-level details of the communication with the daemon.

4.4 APIs and Abstractions

Polycube provides a set of high-level APIs and abstractions to the developers to simplify the writing of a new service, both from the control plane and the data plane point of view. For example, it adds useful abstractions to manage special packets, to cope with special processing that may complicate (and slow down) the fast path, or to react to special events such as timeouts. Polycube implement such functions

Level	Helper Function	Arguments	Description
Fast path (eBPF)	`pcn_pkt_send`	md, out_port	Redirect a pkt to a VNF interface (physical or virtual)
Fast path (eBPF)	`pcn_pkt_controller`	reason	Send a pkt to the slow path with a given reason
Fast path (eBPF)	`pcn_pkt_controller_md`	md, reason	Send a notification to userspace with a specific reason
Fast path (eBPF)	`pcn_call_ingress_program`	index	Call the μCube at a given index attached to the ingress pipeline
Fast path (eBPF)	`pcn_call_egress_program`	index	Call the μCube at a given index attached to the egress pipeline
Slow path (user)	`pcn_packet_in`	pkt, md, reason	Callback executed when a notification is sent to userspace
Slow path (user)	`pcn_send_packet_out`	pkt, dir	Send a packet out to the ingress or egress pipeline
Fast/slow path	`pcn_log`	level, txt	Print debug messages with a given verbosity level
Control plane	`pcn_reload`	code, idx	Reload the μCube at a given index with the new code

Table 4.1: Helper functions provided by Polycube at different level of the NF.

as additional methods loaded and compiled together with the user provided data plane code; then, they will be part of the final object eBPF file that is injected into the kernel. Table 4.1 shows some of the main helper functions introduced by Polycube at different levels of the network service code, i.e., the eBPF fast-path, the slow path and the control and management plane.

4.4.1 Transparent port handling

A Polycube network service instance is composed by a set of virtual ports that are uniquely identified through a *name* and an *index* inside the service itself. Each port of the service can be attached to a Linux network device or to another service port by means of the *peer* parameter. When the fast path of the service decides to redirect the packet to a specific output port it can use the `pcn_pkt_redirect()` function to send the packet to the next hop whether it is a net-device or another Polycube service. Although the implementations for the above two types of next

hops are quite different, Polycube hides this difference by providing a generic helper that receives the virtual index of the output port and, if the port is connected to a netdevice, redirects the packet to the attached netdevice, otherwise jumps directly to the next Polycube network function in the chain.

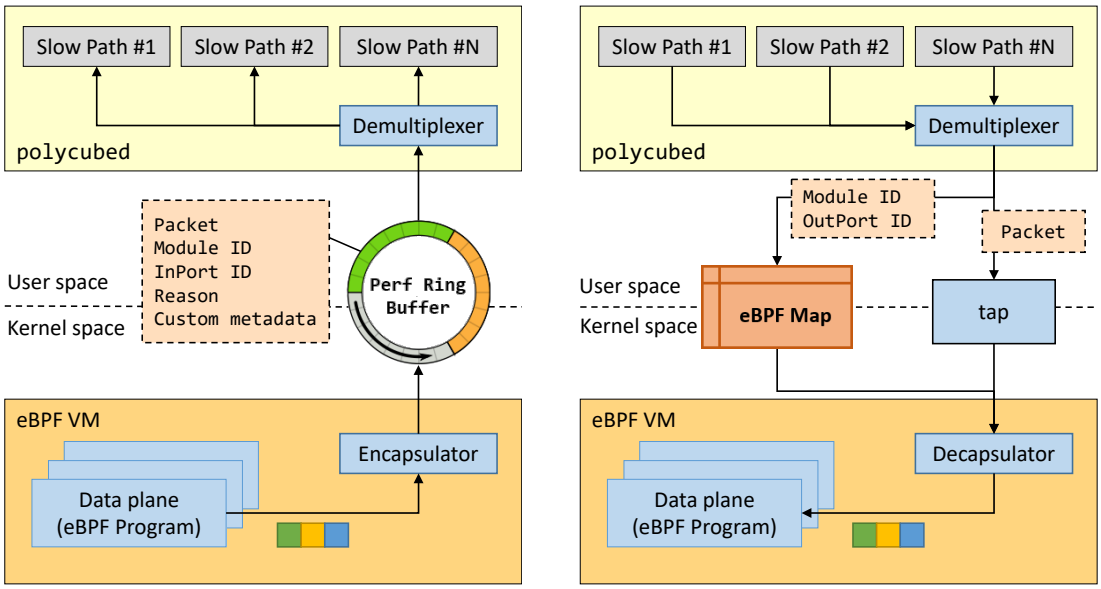
Ports connected to the Linux TCP/IP stack. One of the significant advantages of eBPF is that it is deeply tied to the Linux kernel subsystem, providing the same safety guarantees of the kernel as well as accessing primitives and functionalities available within the core, without having to implement them from scratch. A Cube may want to customize just a small portion of the entire journey of the packet in the Linux TCP/IP stack (e.g., performing rate limiting or DDoS mitigation); in this scenario, after being processed by the Cube the packet should follow the usual processing flow through the networking stack.

Polycube supports this scenario throught the introduction of a special *port* value, called *Host Port*, which connects a port of the Cube directly to the Linux networking stack. When a Cube redirects a packet to this port, both from the control or the data plane, Polycube transparently send it to the host kernel networking stack, where it will continue the standard processing.

4.4.2 Fast-slow path interaction

In Polycube, each instance of a service has its own private copies of *fast* and *slow* paths; Polycube takes care of service isolation by delivering packets generated by the fast path to the corresponding slow path instance and vice versa. It uses two separate (hidden to the developers) eBPF programs, the *Encapsulator* and *Decapsulator*, which are instantiated and injected into the kernel when `polycubed` is started.

Encapsulator. Figure 4.2a shows the flow of operations performed when sending a packet from the eBPF fast path to the slow path running in userspace. If the packet currently processed in the eBPF service fast path requires additional inspections or further processing, it can be sent to the slow path module of the service by means of the `pcn_pkt_redirect_controller()` helper. This function receives as parameters the reason why the packet has to be sent to the slow path and, optionally, additional meta-data fields. Polycube hides the implementation details of the communication between the eBPF fast path program and the service slow path; it sends the packet to an eBPF control module (the *Encapsulator* shown in Figure 4.2a), that will copy the packet and its meta-data into a *perf ring buffer*, which



(a) Encapsulator (b) Decapsulator

Figure 4.2: Message flow for Encapsulator and Decapsulator

is used by `polycubed` to read the corresponding data from userspace[1]. Together with the custom metadata, the *Encapsulator* adds some internal information, such as the *index* of the Cube instance that has generated the message, which are used by the Polycube daemon to call the `packet_in()` function of the associated service's slow path.

Decapsulator. This component handles the reverse communication, which happens when the slow path (or the control plane) of the service wants to inject a packet back in the fast path or send it out on a specific Cube port. In the first case, Polycube simulates the reception of the packet from a specific Cube port, which is specified by the control or slow path module through a specific Polycube helper function, while in the latter case the output port of the Cube is provided. When called, this function triggers the execution of the `Demultiplexer`, which copies the *index* of the Cube originating the message into a specific eBPF map shared with the *Decapsulator*; then, it sends the packet on a `TAP` interface specifically created by `polycubed`. Differently from the `Encapsulator`, it does not use the perf buffer to communicate with the *Demultiplexer*, which is only available for the kernel-userspace communication and not for the opposite. The reception of a packet on the `TAP` interface triggers the execution of the *Decapsulator* eBPF program, which

[1]eBPF provides specific helpers that allow to store custom data into a perf event ring buffer. Userspace programs can then use this buffer as a data channel for receiving events from the kernel.

is attached to the eBPF hook point of the `TAP` interface. When executed, the *Decapsulator* extracts the index of the eBPF program to call from the map and jumps to the next program, following the same operations described in Section 4.5.

4.4.3 Debug mechanism

Polycube provides a debug helper that can be used in both fast and slow/-control path to print debug messages. Although this feature is quite common for userspace programs, it is not the same for the eBPF programs. They can use the `bpf_trace_printk()` function to print debug messages; once the program is loaded, the verifier checks whether program is calling this function and allocates additional buffers, which may slow-down the processing of the function. Polycube uses a more efficient mechanism through the `pcn_log()` helper; when called, this helper uses a *perf ring buffer* to send debug messages to `polycubed`, which redirects them to the current log file, as for the slow and control path. Finally, using different log levels, `polycubed` is able to dynamically remove all the references to the debug messages under the specified log level, reloading the service fast path to reflect the changes.

4.4.4 Table abstractions

To store the network function state across different runs of the same program or to pass configuration data from the control path to the fast path, a Polycube service uses eBPF tables, which are defined into the service fast path and are created when the program is loaded. Every eBPF table has a scope into the system, which expresses the possibility to read and/or modify the table content from another eBPF program. Polycube introduces the possibility to define `PRIVATE` tables, which are only accessible from the same μCube where they have been declared and `PUBLIC` tables, which are instead accessible from every μCube running in the machine. In addition, since Polycube supports the possibility to compose the network function data path as a collection of μCubes (i.e., simple eBPF programs), we added the concept of `SHARED` tables, where a table can in fact be shared between a given set of μCubes. In this case, when the table is instantiated, it is possible to specify the *namespace* within which this table will be shared.

4.4.5 Transparent Support for Multiple Hook Points

Polycube supports two different type of services that corresponds to the existing *networking* "attachments" points (a.k.a., eBPF hooks) available in the eBPF subsystem, namely Traffic Control (TC) and eXpress Data Path (XDP). Although in eBPF, the initial context and the type of operations available differ from the two hooks, Polycube hides those differences allowing the developer to focus only on

the network function semantic, while the framework will take care of handling the differences in the implementation and management of the two subsystems.

Fast path transparency. Polycube provides the ability to write the data plane of a network function (i.e., the eBPF program) independently from the type of hook to which it is attached/ TC and XDP programs differs in: *(i)* the initial context available when the eBPF program is triggered, which represent basically the received packet, *(ii)* the type of helpers that the program is allowed to call and *(iii)* the return type of the program, which communicates the forwarding decision for that specific packet. Polycube hides the above differences providing additional custom helper functions that are used to manipulate or perform different operations on the packets regardless from the type of program currently used. In particular, it "wraps" the execution of the program around two additional components that are executed *before* and *after* a packet enters and/or leaves the Cube. Those wrappers take care of converting the original context into a standard Polycube format and perform the reverse translation on the opposite direction. Note: we will see in Section 4.5 how these wrappers are fundamental for the implementation of the Polycube service chaining.

Slow & Control path transparency. Even the interaction between the fast and slow path is subjected to differences depending on the type of hook point in which the function is executed. As for the previous case, Polycube provides a transparent API for this interaction, hiding the details to the developers, who do not have to worry about the mechanisms used to exchange data. The *encapsulator* and *decapsulator*, shown in Section 4.4.2 are another example of this seamlessy interaction. The implementation of those components depend on the hook point at which the service is attached; when a packet from the control/slow path of a service is injected into the fast path, `polycubed` selects the appropriate control program depending on the type of the service instance.

4.4.6 Transparent Services

A Polycube standard service (i.e., a Cube) is made of a set of virtual ports. When a packet is received on a specific port, the Cube takes a forwarding decision to redirect the packet to one of its output interfaces. This process may depend on the specific port configuration and the behavior configured for each service. For example, a *router* Cube has an IP address associated to each of its ports; when a new packet arrives it checks the routing table to find the next-hop address and forward the packet consequently.

On the other hand, there are services that do not have any specific port configuration, and their behavior is independent from the number of ports or their parameters. To address this case, Polycube introduces the concept of *transparent*

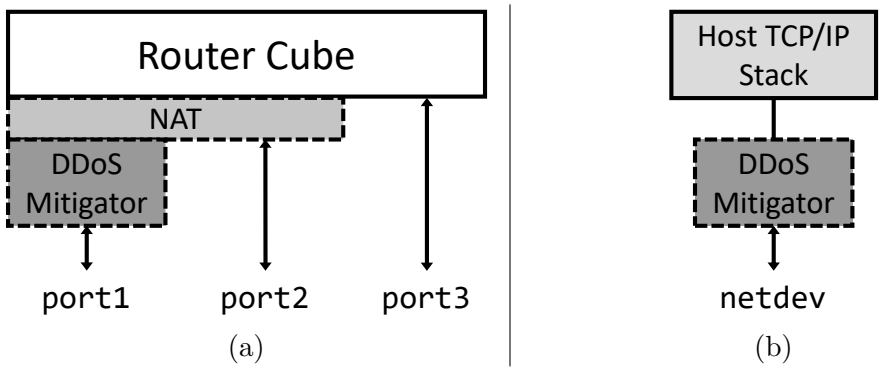


Figure 4.3: (a) Transparent cubes attached to a port of the service. (b) Transparent cube attached to netdev.

cubes. A transparent cube does not take any forwarding decision. It is composed of an *ingress* pipeline that is called when a packet enters the service, and an *egress* pipeline that is called when the packet leaves the cube. Moreover, it has to be attached to a specific port that can be either a virtual port of another cube or a Linux netdevice.

Transparent services attached to Cube's port. Multiple transparent cubes can be attached to a port of a Cube by specifying a position in which the transparent service should be placed with respect to the others transparent cubes; this also defines the order of execution of the services. When attached, they have the possibility to inherit some specific parent's port configurations that can be used to automatically configure the service itself. For example, in Figure 4.3a a NAT service attached to a router's port can read the corresponding IP address and use it for the NAT translation.

Transparent services attached to netdevs. A transparent cube can be also directly attached to a Linux netdev, as shown in Figure 4.3b. When a packet reaches the netdevice, the *ingress* pipeline of the service is called; once completed, the packet is passed to the host's TCP/IP stack. At the same way, when a packet is sent out to the netdevice, the *egress* pipeline is executed. Polycube loads the ingress pipeline into the eBPF ingress hook (i.e., XDP or TC_INGRESS) and egress (i.e., TC_EGRESS) hooks.

4.5 Service Chaining Design

A Polycube service chain involves of a set of network function instances that are connected to each other by means of virtual ports, which are in turn peered with a Linux networking device or another in-kernel NF instance. In the standard model, eBPF programs do not have the concept of port from which traffic is received or sent out; it only provides a *tail call* mechanism to "jump" from one program to another.

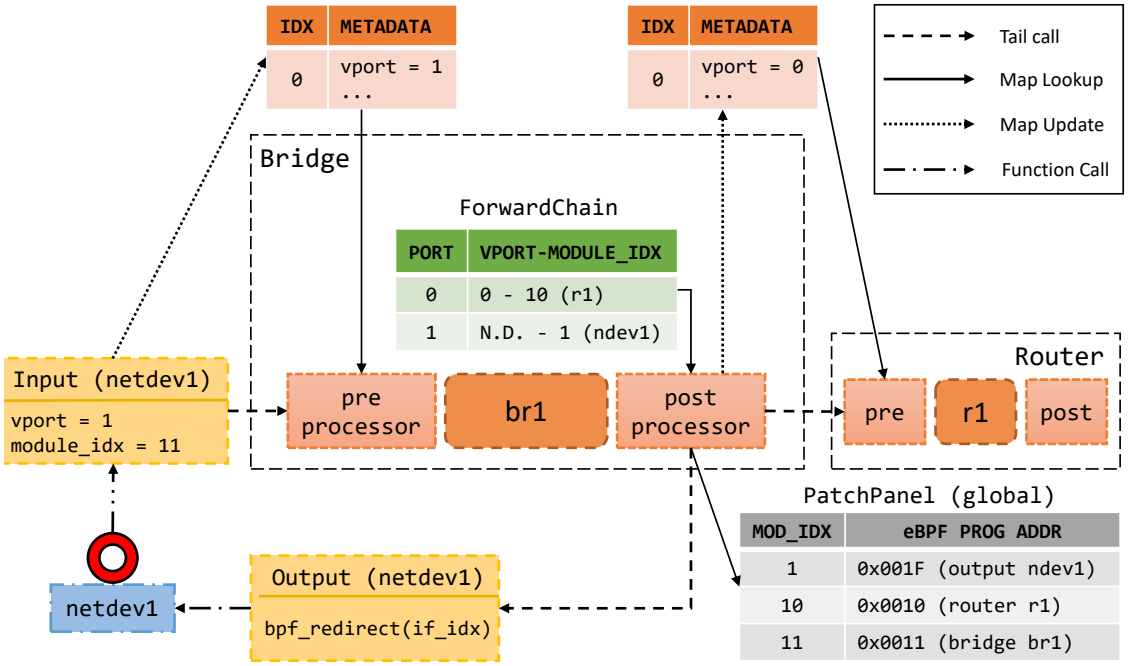


Figure 4.4: Internal details of the Polycube service chains

To provide this abstraction, Polycube uses a set of additional eBPF components and wrappers around the user-defined code; Figure 4.4 shows the resulting design.

When a packet traverses a chain in Polycube, it carries some metadata (e.g., ingress virtual port, module index), which are internally used by Polycube to correctly isolate the various NFs and implement the desired chain. In particular, the *cube index* is used to uniquely identify the Cube fast path inside the framework and it is uniquely generated when the Cube instance is created. Before injecting the Cube's fast path, Polycube augments the user-defined code with a set of *wrappers* that are executed *before* and *after* the service itself. In particular, the *pre-processor* contains the set of functions necessary to process the incoming traffic, while the *post-processor* contains the helpers used by the fast path to send the traffic outside of the Cube.

Pipeline Example. We will now walk through a simple example to illustrate how packets are passed through the Polycube service chain. In Figure 4.4 we have an instance of a Polycube Bridge service, called `br1` with two virtual ports connected respectively to a Linux networking device (i.e., `netdev1`) and to the first port of an instance of *Router* Cube, called `r1`. First, when a new port is created in the `br1` instance, Polycube assigns a unique virtual port identifier (i.e., *vport*) to each port; in our case, the port attached to the router has id #0, while the other has

id #1. At the same way, the router's port attached to the bridge has id #0. When a new packet is received to the physical interface `netdev1`, it has to execute the `br1` fast path and be presented as coming from the virtual port #1. To support this abstraction, every time a Cube port is peered with a Linux networking device, Polycube loads two additional eBPF programs. The `Input` eBPF program, which is attached to the ingress hook of the interface, is loaded and compiled at runtime with some pre-defined information such as the *virtual port id* (i.e., #1 in the example in the figure) associated by Polycube to that Cube port and the `index` of the module to which the port is attached (i.e., #11). Upon the reception of a new packet from the physical interface attached to the bridge, the `input` program is triggered; it copies the *vport* value into an eBPF per-cpu array map shared with the bridge instance and performs a tail call to the `br1` pre-processor, using the hard-coded module index[2].

At this point, the control passes to the *pre-processor* module of `br1`, which extracts the `vport` from the shared map (and possible additional metadata such as the packet length, headers) and invokes the `br1` code. The Cube fast path can then use the `vport` to send traffic outside as result of a forwarding decision; this is possible through a specific helper function contained in the *post-processor*, which will redirect the packet to the next module of the chain. The *post-processor* uses an additional auxiliary data structure, the `ForwardChain`, to obtain the index of the next module of the chain corresponding to the given vport. This map has a local scope and represents the actual connection matrix of the service instance with the rest of the world. In our example, the lookup into the `ForwardChain` map with vport #0 returns the next virtual port id of `r1` (i.e., `0`) and the index of the eBPF program corresponding to that Cube (i.e., `10`). As before, the *post-processor* copies the vport into the shared map and "jumps" to the `r1` pre-processor, which can then perform a tail call using the `PatchPanel` to get the real address of the next eBPF program. On the other hand, if `br1` decides to redirect the packet to the port #0, the *post-processor* retrieves the next module index (i.e., `1`) from the `ForwardChain` and jumps to this module, which corresponds to the *Output* program associated with the physical interface. As for the *input* program, this module is injected with a pre-defined `if_index` of the netdevice, which uses in the `bpf_redirect()` helper function to send the packet out on `netdev1`.

To simplify the pipeline example, we have omitted the case in which the Cube is composed of several μCubes (section 4.3.2.1). Conceptually, the operations remain the same; the only difference is that, after the *pre-processor*, the code of the *MASTER* μCube is called, which in turn uses an internal *ForwardChain* to jump from one μCube to another.

[2]The only way to share information from one eBPF program to another is to copy the data into shared eBPF maps. For the internal communications, Polycube uses per-cpu maps, which provide better performance thanks to their lockless access. Since a packet is processed only within a single core (eBPF does not allow *preemption*), this mechanism is safe.

4.6 Management and Control Plane

The capability to add (or remove) a network function dynamically (even from a remote server) into `polycubed` provides several advantages such as the possibility to update an existing service, adding functionality without modifying the network functions currently deployed and running. To support this model, the Polycube core (i.e., `polycubed`) has been designed to be completely independent from the type of network function that is installed. `Polycubed` has no idea of how the network function is composed internally or what are its functionalities, and it takes only care of forwarding the request to the proper service instance. This approach does not require changes to `polycubed` whenever changes to the individual service are needed; when the service is being updated, it is unplugged from the framework, updated and plugged-in again without affecting existing services. On the other hand, it complicates the service design, which has to define the interface to the outside (i.e., the `REST` APIs). To simplify this process, Polycube uses `YANG` [25] models, each one describing a specific service, to automatically synthesize the `REST` interface of the service.

4.6.1 Model-driven service abstraction

The `YANG` data modeling language allows to *(i)* model the structure of the data and the functionalities provided by the Polycube service, *(ii)* define the semantic of the service data and their relationship and *(iii)* express their syntax, which will be used to interact with the service itself. When a new service is registered, `polycubed` reads the provided `YANG` model and generates an internal representation of the service data together with a specific path mapping table used to access those data from outside. Whenever a new request for that service arrives, `polycubed` validates it (e.g., checking the correct format of an IP address, ports in a given range) according to the information specified in the `YANG` model, without having to rely on the service itself for those "ancillary" tasks.

4.6.1.1 Service base data model

Polycube provides to the developers a basic service structure that can be extended to compose the desired network function, offering fundamental abstractions (e.g., VNF ports, port peers, hook type) that are used to simplify the interaction between the different services and system components. The basic structure, shown in the YANG Listing 4.1, reflects the internal representation of a Polycube service, with its primary parameters and components. Every service within the framework is uniquely identified through a name, which is specified using the `service-name` extension; Polycube does not allow multiple services with the same name. The

```
module pcn-base-service-model {
  extension service-description {...}
  extension service-version {...}
  extension service-name {...}
  extension service-min-kernel-version {}

  grouping base-service-instance {
    leaf name {...}
    leaf uuid {...}
    leaf loglevel {...}
    leaf hook {
      type enumeration {
        enum TYPE_TC;
        enum TYPE_XDP_SKB;
        enum TYPE_XDP_DRV;
      }
    }
    list ports {
      key "name";
      unique "uuid";

      leaf name {...}
      leaf uuid {...}
      leaf peer {...}
    }
  }
}
```

Listing 4.1: The *base* YANG model of a Cube

`service-min-kernel-version` is used to indicate the minimum kernel version required to execute the service, since there are some eBPF functionalities that are available only on newer kernel versions; when the service is loaded, Polycube checks if the host is running a kernel version greater than or equal to this value. Finally, the `service-description` and `service-version` are used to describe the current service. While the previously mentioned information describe the service itself (e.g., a firewall Polycube NF), the variables under the *grouping* statement are specific for each service instance (e.g., a firewall *fw1*). Each instance is identified with a `name`, which is unique inside the service scope, the `hook` point at which the instance is attached to, and a list of `ports`, identified with a unique *name* inside the service instance and a *peer*.

4.6.1.2 Automatic REST API generation

Polycube uses the information in the YANG model to automatically derive the set of REST APIs that are used to interact with the service. As shown in

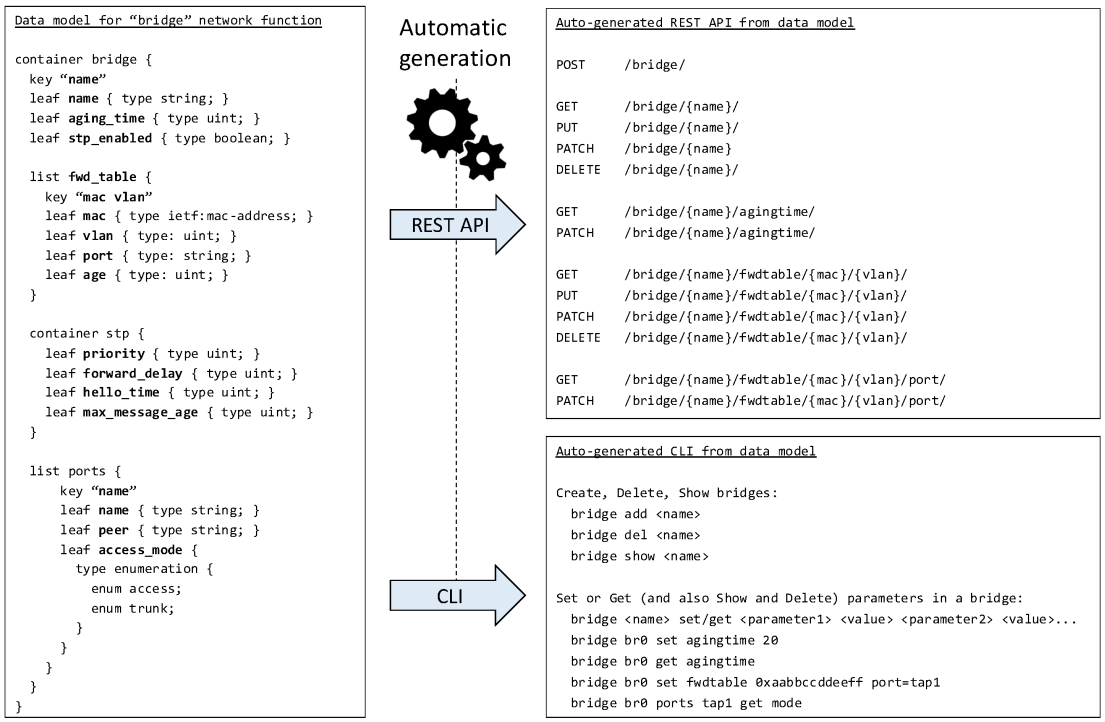


Figure 4.5: YANG to REST/CLI service description

Figure 4.5, each YANG resource is automatically mapped to a specific URL, while the different HTTP methods are used to identify the operations required for a particular resource. The `GET` operation allows to obtain the current value of a given resource on the Polycube service, the `POST` operation is used to create an instance of the resource, the `PATCH` operation is used to modify the current value of the resource and finally, the `DELETE` operation is used to delete the specified resource. This approach is similar to the one adopted by the RESTCONF specification [24], which aims at providing a programmatic interface for accessing data defined in YANG, allowing any client to communicate with the Polycube service by just knowing its YANG module.

4.7 Implementation

4.7.1 Polycube Core

As of today, the code of Polycube, i.e., `polycubed` is implemented in 28k lines of C++ code, running within an unmodified Linux without having to install custom drivers or specific kernel modules. It only requires a v4.15 as minimum kernel version to run the deamon; then, every specific service may have its own requirements

depending on the functionalities that they use. The Polycube daemon contains both the code required to handle the different Polycube NFs but also the service-agnostic server proxies, which parses at runtime the YANG model of every loaded service to generate the appropriate REST API and to perform the validation of the service parameters within the server itself.

Network Function	**Minim. kernel**	**LoC FP**	**LoC SP**	**LoC CP (AU/-MAN)**	**Description**
Bridge	4.15	239	40	5798 / 1105	A L2 switch NF with support for VLAN and STP.
DDoS Mitigator	4.15	140	/	1850 / 20	A NF that drops (malicious) packets based on a blacklist applied on either IP src and dst addresses.
Firewall	4.19	1654	/	5951 / 2945	A firewall service that drops or allows packets based on the configured rules.
Dynamic Monitor	4.15	/	/	2330 / 673	Generic service used to inject eBPF code that monitors network traffic, collecting and exporting custom metrics. The data plane size depends on the injected code.
Load Balancer (DSR)	4.15	362	/	3476 / 566	A version of the Maglev scalable load-balancer [57].
Packet Capture	4.15	/	/	2511 / 822	A NF use to capture packets flowing through a Linux netdevice or between other cubes.
Policy-Based Forwarder	4.15	243	/	2605 / 240	A simple ACL-based forwarder.
Router	4.15	276	120	3168 / 1030	A router NF.
NAT	4.15	380	/	6302 / 208	A NF that support Source, Masquerade, Destination NAT and Port Forwarding.
Iptables	5.3	2254	/	6409 / 1787	Special service that emulates the behavior of iptables [115].

Table 4.2: A list of NF implemented with Polycube.

Polycube is built around the BPF Compiler Collection (BCC) [17], which provides a set of abstractions to interact with eBPF data structure or to load/unload eBPF programs, together with a compilation toolchain that include Clang/LLVM to allow a dynamic generation of the eBPF code that is injected in the kernel. Polycube extends those abstractions with additional helper functions targeted to networking services and to the Polycube NF structure. In particular, the availability of the compilation toolchain allows to re-compile the code at runtime, enabling more aggressive service optimizations that can be dynamically applied within the NF, as we will see in the next Chapters.

4.7.2 Polycube Services

To stress test the generality of the Polycube programming model and abstractions, we have implemented a large range of network functions. Table 4.2 briefly describes the type, role of each of them and the lines of code (LoC) required to implement the fast path (FP), the slow path (SP) and the control path (CP). In the former parameter we distinguish the LoC automatically generated from the YANG model (i.e., CP/AU) and the one manually written (i.e., CP/MAN).

At the time writing, there are 18 different services implemented in Polycube, with an overall number of about 54k lines of code, which include both the C++/C code of the control and slow path and the eBPF code of the fast path. These implementations suggest that Polycube succeed in the role of providing a generic and highly customizable framework that can be used to implement a wide variety of NFs.

4.7.2.1 Example: L2 Switch

Control Plane. In Listing 4.2 we show a small code snippet of the control plane (CP) of the Polycube L2 Switch NF, in order to show the Polycube programming model and abstraction provided.

```
1  Switch(const std::string name, const SwitchJsonObject &conf)
2      : Cube(dp_code, conf.getCubeType()) {
3    logger()->info("Creating Switch instance");
4
5    stp_enabled_ = conf.getStpEnabled();
6
7    addPortsList(conf.getPorts());
8    insertFdb(conf.getFdb());
9    updateStpList(conf.getStp());
10 }
```

Listing 4.2: Sample CP code from Polycube L2 Switch NF.

When a new Switch Cube instance is requested to `polycubed` (via CLI or REST API), the constructor is of the Switch object is automatically called, receiving the

`name` of the Cube together with a configuration object that contains all the parameters specified in the YANG model and automatically validated by `polycubed`. As shown in line #2, the constructor provides the data plane code (i.e., the eBPF code) to the Cube object, together with its type, which corresponds to the attachment point of the cube itself. Then, Polycube will take care of compiling and loading the code in the appropriate point of the NF chain.

When new request to the control plane arrives, other methods of the Switch object are asynchronously called, allowing the user to implement the desired behavior without having to handle the single REST API call. Listing 4.3 show an example of those methods. The stub of these methods is generated automatically from the YANG model; then, the user has to fill the corresponding methods with the specific implementation. In line #8 and #36 we can notice how Polycube provides also abstraction to interact with the eBPF data plane. In this case, it can retrieve the map `fwdtable` defined in the eBPF code and interact with it to *set* or *get* a specific entry.

```
1  std::shared_ptr<FdbEntry> Fdb::getEntry(const uint16_t &vlan,
2                                          const std::string &mac) {
3    std::lock_guard<std::mutex> guard(fdb_mutex_);
4
5    logger()->debug("[Fdb] Received request to read map entry");
6    logger()->debug("[Fdb] vlan: {0} mac: {1}", vlan, mac);
7
8    auto fwdtable = get_hash_table<fwd_key, fwd_entry>("fwdtable");
9    fwd_key key{
10       .vlan = vlan,
11       .mac = polycube::service::utils::mac_string_to_nbo_uint(mac),
12   };
13
14   return fwdtable.get(key);
15
16   logger()->debug("[Fdb] Entry read successfully");
17 }
18 ...
19
20 void Fdb::addEntry(const uint16_t &vlan, const std::string &mac,
21                    const FdbEntryJsonObject &conf) {
22   std::lock_guard<std::mutex> guard(fdb_mutex_);
23
24   fwd_key key{
25       .vlan = vlan,
26       .mac = polycube::service::utils::mac_string_to_nbo_uint(mac),
27   };
28
29   fwd_entry value{
30       .timestamp = timestamp,
31       .port = port_index,
32       .type = STATIC,
```

```
33      };
34
35      auto fwdtable = get_hash_table<fwd_key, fwd_entry>("fwdtable");
36      fwdtable.set(key, value);
37      logger()->debug("[Fdb] Entry inserted");
38  }
```

Listing 4.3: Automatically generated methods of the Polycube L2 Switch NF.

Finally, Listing 4.4 shows the `packet_in` method that is asynchronously called when a packet is sent from the DP to the CP. This method receives the packet itself and a set of metadata associated with it, that can be used to understand the reason of the CP intervention. Finally, when the packet is processed, as shown in line #13, the packet can be sent out to a specific port.

```
1   void Switch::packet_in(Ports &port, PacketInMetadata &md,
2                          const std::vector<uint8_t> &packet) {
3     logger()->debug("Packet received from port {0}", port.name());
4     switch (static_cast<SlowPathReason>(md.reason)) {
5       case SlowPathReason::BROADCAST:
6         broadcastPacket(port, md, packet);
7         break;
8       case SlowPathReason::BPDU:
9         if (stp_enabled_)
10          processBPDU(port, md, packet);
11        else
12          broadcastPacket(port, md, packet);
13          port.send_packet_out(packet, tagged, vlan);
14        break;
15      default:
16        logger()->error("Not valid reason {0} received", md.reason);
17      }
18  }
```

Listing 4.4: Packet-in method called when a packet is sent from the DP to the CP.

Data Plane. At the same way, Listing 4.5 shows a code snippet of the Polycube L2 Switch NF data plane. The ingress point of the NF is the `handle_rx` method, which is called every time a new packet enters this NF. This method is called after the internal (and hidden) Polycube functions, whose behavior is to simulate the NF chain and fill the appropriate helper variable such as the `pkt_metadata` structure, as explained in Section 4.5.

When the processing is completed, the DP can use the Polycube specific helper to redirect a packet to a specific NF interface (e.g., line #59), following is journey through the Polycube NF chain or *drop* the packet, as shown in line #56. On the other hand, if the packet requires further processing, it can be sent to the control plane with a specific *reason*, as shown in line #63. Finally, it is worth to note that

the `pcn_log()` functions can be used to print debug messages with specific levels of debug and they are automatically compiled in or out by Polycube without any user intervention.

```
1
2  int handle_rx(struct CTXTYPE *ctx, struct pkt_metadata *md) {
3    struct eth_hdr *eth = data;
4
5    if (data + sizeof(*eth) > data_end)
6      return RX_DROP;
7
8    u32 in_ifc = md->in_port;
9
10   pcn_log(ctx, LOG_TRACE, "New packet from port %d", in_ifc);
11
12   // LEARNING PHASE
13   __be64 src_key = eth->src;
14   u32 now = time_get_sec();
15
16   struct fwd_entry *entry = fwdtable.lookup(&src_key);
17
18   if (!entry) {
19     struct fwd_entry e;  // used to update the entry in the fdb
20
21     e.timestamp = now;
22     e.port = in_ifc;
23
24     fwdtable.update(&src_key, &e);
25     pcn_log(ctx, LOG_TRACE, "MAC: %M learned", src_key);
26   } else {
27     entry->port = in_ifc;
28     entry->timestamp = now;
29   }
30
31   // FORWARDING PHASE: select interface(s) to send the packet
32   __be64 dst_mac = eth->dst;
33   entry = fwdtable.lookup(&dst_mac);
34   if (!entry) {
35     pcn_log(ctx, LOG_DEBUG, "Entry not found");
36     goto DO_FLOODING;
37   }
38
39   u64 timestamp = entry->timestamp;
40
41   // Check if the entry is still valid (not too old)
42   if ((now - timestamp) > FDB_TIMEOUT) {
43     pcn_log(ctx, LOG_TRACE, "Entry is too old. FLOODING");
44     fwdtable.delete(&dst_mac);
45     goto DO_FLOODING;
46   }
```

```
47
48    pcn_log(ctx, LOG_TRACE, "Entry is valid. FORWARDING");
49
50  FORWARD:;
51    u32 dst_interface = entry->port;
52
53    // HIT in forwarding table
54    /* do not send packet back on the ingress interface */
55    if (dst_interface == in_ifc) {
56      return RX_DROP;
57    }
58
59    return pcn_pkt_redirect(ctx, md, dst_interface);
60
61  DO_FLOODING:
62    pcn_log(ctx, LOG_DEBUG, "Send to controller");
63    pcn_pkt_controller(ctx, md, REASON_FLOODING);
64    return RX_DROP;
65  }
```

Listing 4.5: DP of the Polycube L2 Switch NF.

4.8 Evaluation

In this section we evaluate the performance of a set of Polycube NFs and we compare the results with existing in-kernel implementations. Fist, we measure the performance of standalone Polycube NFs and then we move to more complex scenarios where chains of NFs are involved (section 4.8.2). Finally, we evaluate the overhead imposed by Polycube programming model when compared to baseline programs written using the vanilla eBPF (section 4.8.3).

4.8.1 Setup

We run our experiments into a server equipped with an Intel Xeon Gold 5120 14-cores CPU @2.20GHz (hyper-threading disabled) 19.25 MB of L3 cache and two 32GB RAM modules. The packet generator is equipped with an Intel Xeon CPU E3-1245 v5 4-cores CPU @3.50GHz (8 cores with hyper-threading), 8MB of L3 cache and two 16GB RAM modules. We used Pktgen-DPDK [55] to generate 64-bytes UDP packets and to count the received packets. In fact, each server has a dual-port Intel XL710 40Gbps NIC, directly connected to the corresponding one of the other server. Both servers run Ubuntu 18.04.1 LTS, with the DUT running kernel v5.6 and the eBPF JIT flag enabled (the default behavior for newer kernels).

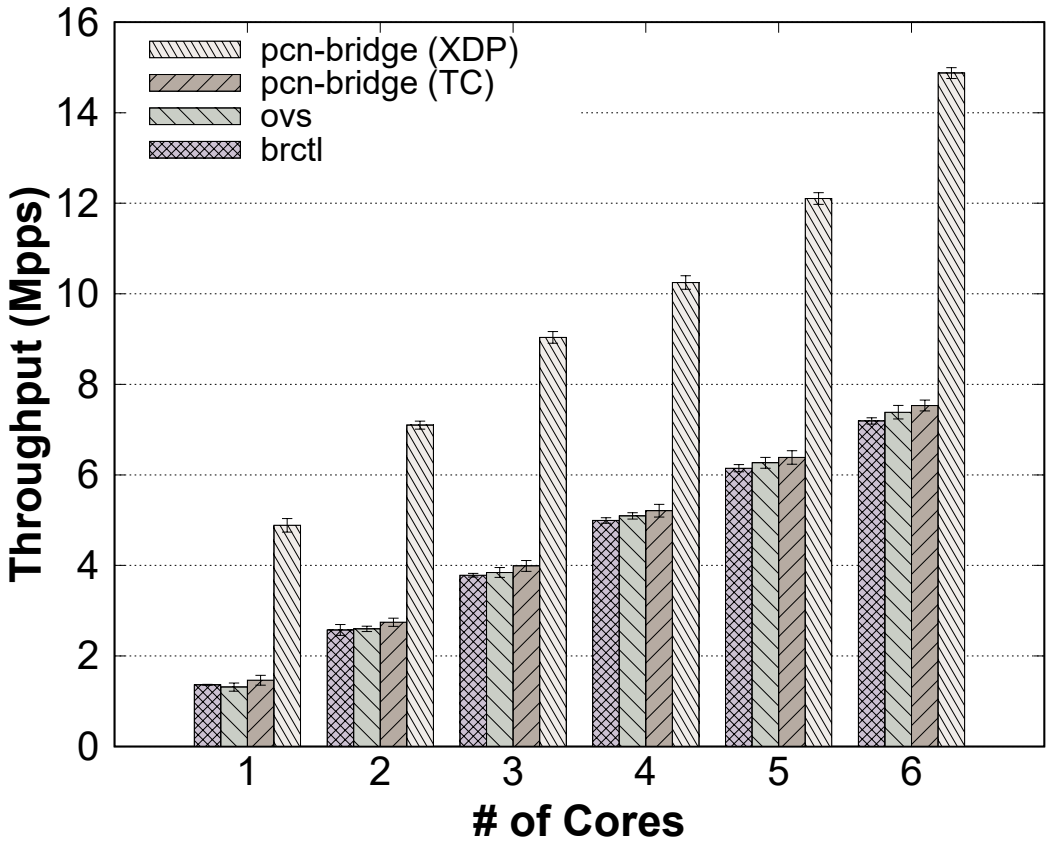


Figure 4.6: Packet forwarding throughput comparison between Polycube *pcn-bridge* NF (in both XDP and TC mode) and "standard" Linux implementation such as Linux bridge (*btctl*) and OpenvSwitch (*ovs*).

4.8.2 Test Applications

4.8.2.1 Case Study 1: L2 Switch

In this test scenario, we evaluate the performance of a Polycube NF that emulates the behavior of a fully functional L2 switch with support for VLAN and Spanning Tree Protocol (STP). The *pcn-bridge* data plane is implemented entirely in eBPF, including the MAC Learning phase. More complex functionalities such as the handling of STP protocol BPDUs or flooding, as results of a miss in the filtering database, are relegated to the slow-path, given the impossibility of performing such actions entirely in eBPF. We measure the UDP forwarding performance between *pcn-bridge* and commonly used L2 switch Linux tools such as Linux bridge (v1.5)

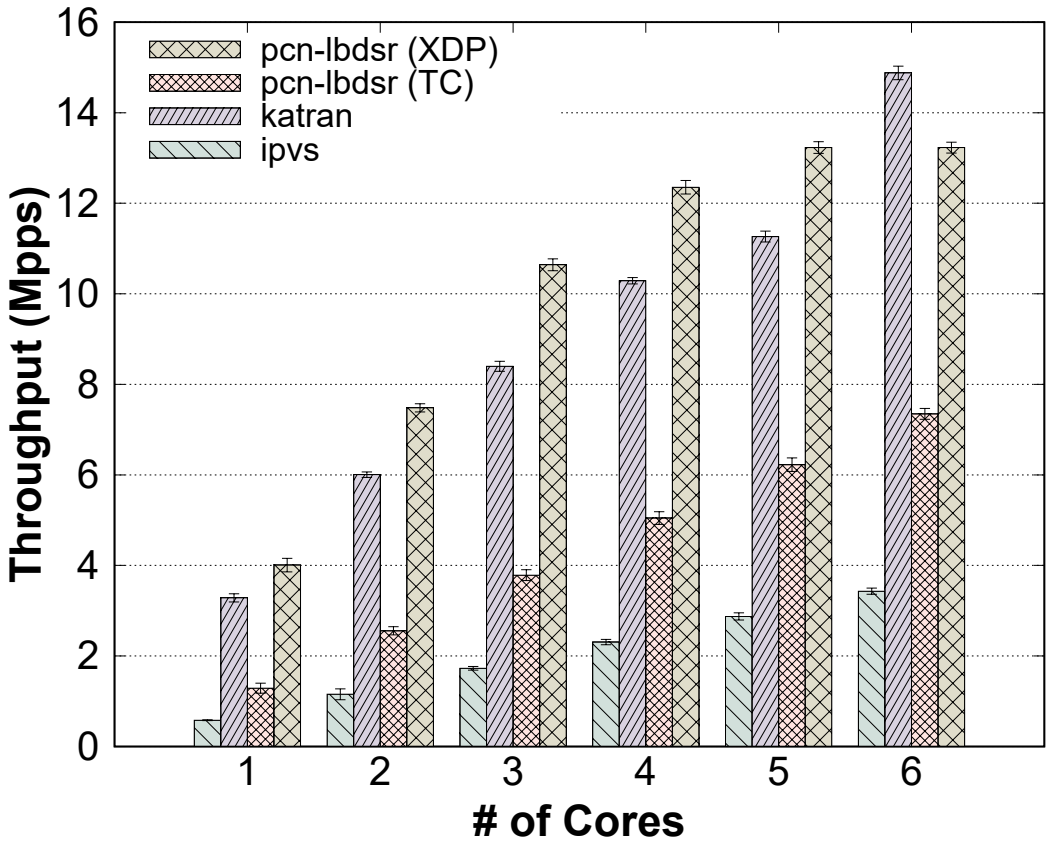


Figure 4.7: Throughput performance between a Polycube load balancer NF (i.e., *pcn-lbdsr*), *ipvs*, the standard L4 load balancing software inside the Linux kernel and *Katran*, an XDP-based load balancer developed by Facebook.

and OpenvSwitch (v2.13) [3]. We can clearly notice, from Figure **??**, that *pcn-bridge* outperforms the other tools in both TC and XDP mode, with a performance gain of about 3.6x for the latter. The advantages of XDP are more evident since packets are processed directly at driver level, avoiding the overhead given by the allocation of kernel data structures, which may be unnecessary for the simple forwarding use case. Moreover, we can notice how the performance of our *pcn-bridge* NF scale linearly with the number of cores used, reaching 10Gbps throughput with 64B packets with six cores involved[4].

[3]All the tests with OpenvSwitch have been carried out on kernel v5.0 since it does not support earlier versions.

[4]To gradually use all cores, we have configured the hardware filtering on the NIC (i.e., Flow Director rules) to redirect different flows to distinct NIC's RX queues.

4.8.2.2 Case Study 2: Load Balancer

In this test, we measure the performance of a Polycube load balancer NF (i.e., *pcn-lbdsr*). As for the previous scenario, this NF is implemented as a single μCube, with the data plane entirely handled in eBPF (no slow-path is involved). Polycube *pcn-lbdsr* can be configured with a list of virtual IPs (VIP), each one with an associated list of back-ends; we use the Maglev [57] hash to select the back-end server, which provides a better resilience to back-end server failures, a better distribution of the traffic load among the back-ends and the possibility to set different weights for each back-end server. To test this scenario, we used a fixed number of hosts[5], as we compared the throughput results with IPVS v1.28 and Katran [76], an eBPF/XDP-based load-balancer developed and released as open-source by Facebook. Figure 4.7 shows the results of this test, with both *pcn-lbdsr* in TC and XDP mode that outperform *ipvs* by a factor of 2.2x and 6.5x respectively. Moreover, we want to notice that *pcn-lbdsr* in XDP mode offers performance comparable (or even higher) with *Katran*. This results is mainly given by the heavy use the Polycube NFs make of the dynamic reloading feature, which allows the NF to better adapt to the runtime configuration by compiling out features that are not required at runtime, hence, improving the overall data plane performance. The other big difference is that *Katran* is a standalone application built only for the load-balancing use case; on the other hand, Polycube offers a general framework to build and create complex NF chains, allowing to create more complex network typologies, while still providing performance comparable with "native" eBPF implementations.

4.8.2.3 Case Study 3: Firewall

The Polycube *pcn-firewall* NF is implemented as a series a μCubes that compose the entire firewall pipeline (even in this case, the slow path is not involved). It implements the Linear Bit Vector Search (LBVS) [98] classification algorithm to filter packets, with a sequence of μCubes each one in charge of handling specific fields of the packet (e.g., IP source, destination, protocol, etc.)[6]. Moreover, it is also able to "communicate" with the Linux routing table to check the next hop address before forwarding a packet to the egress interface[7]. For this test we used a synthetic ruleset generated by classbench [155], with all the rules matching the TCP/IP 5-tuple. The default rule of the firewall is to *drop* all the traffic, while

[5]We used the same configuration of [75] for the load balancer use case, setting one virtual IP per CPU core and 100 back-end servers for each VIP.

[6]A more detailed explanation of the Polycube firewall architecture is provided in [115], whose data and control plane has been implemented as a standalone Polycube network service.

[7]eBPF provides a specific helper that can be used to lookup the FIB Linux table directly from the XDP code.

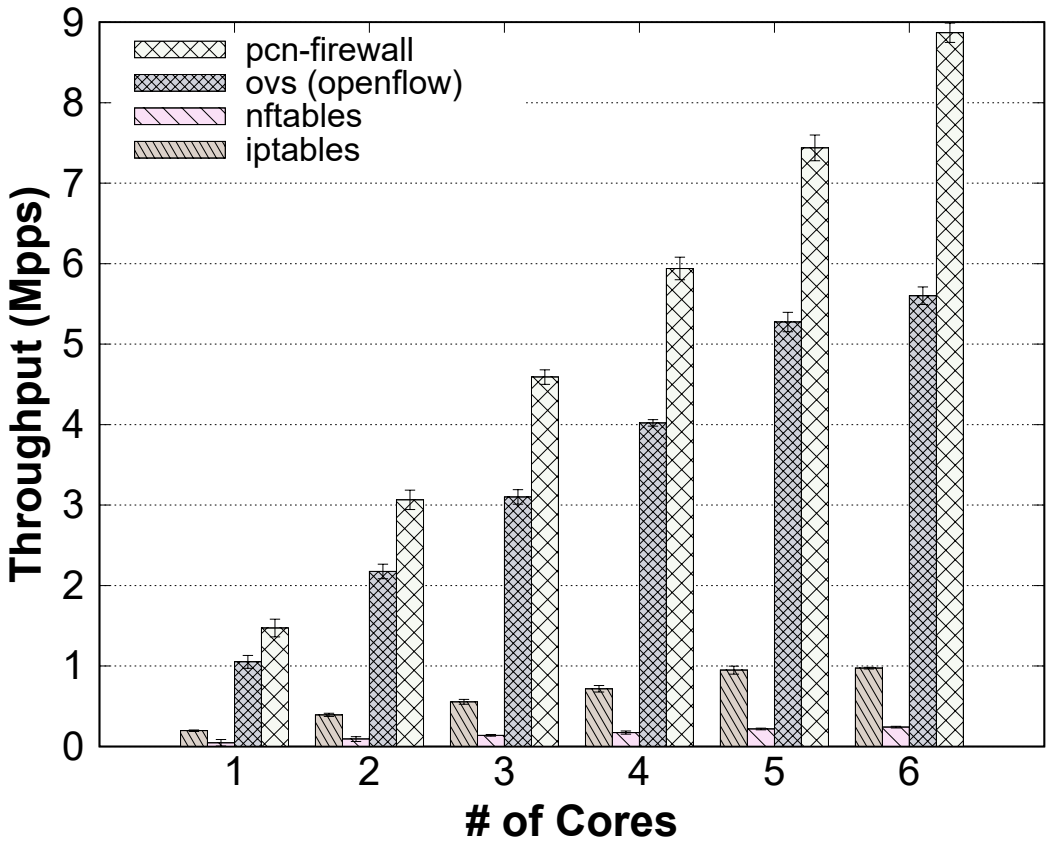


Figure 4.8: Throughput performance comparing with 1000 rules between a Polycube firewall NF (i.e., *pcn-firewall*), *iptables* and *nftables*, which are two commonly used Linux firewalls and OpenvSwitch (*ovs*) with OpenFlow rules.

only the matching flows are "allowed" and redirected to the second interface of the DUT. The generated traffic is uniformly distributed among all the rules so that all generated packets should be forwarded.

We compare the performance of *pcn-firewall* with both *iptables* and *nftables*, the most used packet filtering software used in the Linux subsystem today. Then, we also load the same ruleset as a set of OpenFlow rules in the OpenvSwitch pipeline[8] and we measure the performance under the same conditions mentioned before. The results are shown in Figure 4.8. Even in this case, we can clearly see how *pcn-firewall* outperforms the existing solutions by a 31.8x, 7.5x and 1.4x factor respectively for *nftables*, *iptables* and *ovs*. The reason of this is twofold. First, *pcn-firewall* implements a faster classification algorithm compare to the linear scanning implemented by Linux-native firewalls and second, it can adopt more aggressive

[8]To generate the OpenFlow rules we used Classbench-ng [**7966918**].

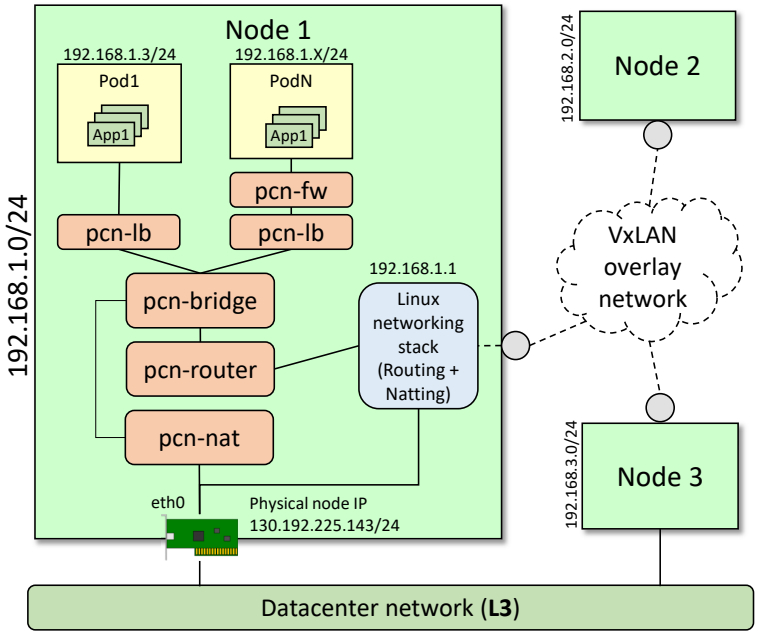


Figure 4.9: Architecture of the Polycube K8s plugin.

optimizations thanks to the dynamic reloading feature of Polycube, allowing the control plane to specialize the packet processing behavior depending on the actual firewall configuration (e.g., the deployed ruleset).

4.8.2.4 Case Study 4: K8s Network Provider

To demonstrate the capability of Polycube to enable the creation of complex applications created by chaining different network functions together, we present a real world use case that can be implemented within Polycube and the type of performance improvements that we can expect. In particular, we implemented a CNI plugin for Kubernetes [82], one of the most important open source orchestration system for containerized applications. A K8s network provider must implement Pod-to-Pod communication[9], provide support for ClusterIP services[10] and security policies; our prototype supports all of them. Our overall design includes the five different components: a NAT NF (*pcn-nat*), a L3 routing module (*pcn-router*), a

[9]A Pod is the smallest manageable unit in a k8s cluster and is composed of a group of one or more containers sharing the same network.

[10]A ClusterIP is a type of service that is only accessible within a Kubernetes cluster through a *virtual* IP. When a Pod communicates with this *virtual* IP, the request can be mapped to an arbitrary Pod running within the same physical host or into another one.

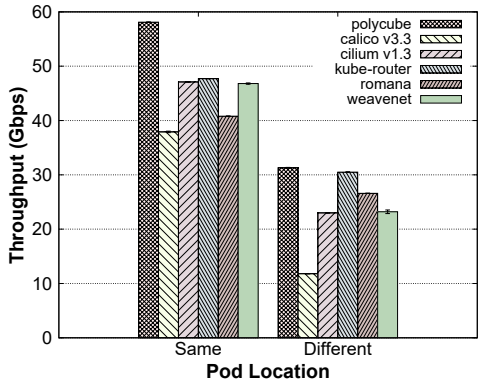


Figure 4.10: Performance of different k8s network providers for direct Pod to Pod communication.

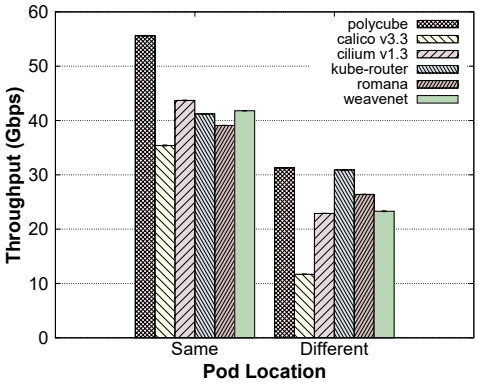


Figure 4.11: Performance of different k8s network providers for Pod to ClusterIP communication.

L2 switch application (*pcn-bridge*), a load balancer (*pcn-lb*) and a firewall component (*pcn-firewall*). These NFs are chained together by means of Polycube APIs, as shown in Figure 4.9, and are configured to support the main operations required by the k8s network plugin interface. Tests were carried out on a 3-node cluster, a master and two workers with Linux kernel v4.15, Intel Xeon CPU E3-1245v5 @3.50GHz with dual-port Intel XL710 40Gbps NIC cards connected point-to-point. We report the TCP throughput measured with `iperf3` using the default parameters; the server was always running in a Pod, while the client was either in a physical machine or in another Pod depending on the test.

In Figure 4.10 and 4.11 we asses the performance of our Polycube network provider compared to other existing solutions. In particular, we consider the Pod-to-Pod connectivity and the Pod-to-ClusterIP connectivity. Results show that the Polycube k8s plugin reaches 15-20% higher throughput than other solutions in the case server and client are on the same node. When pods are on different nodes, the advantage of the plugin becomes less evident because of the influence of the physical network, but still better than other solutions. Indeed, those providers often relies on existing kernel components, such as *iptables* and Linux bridge, that we proved in the previous case to be less efficient than the Polycube counterparts.

Although our k8s plugin achieves better performance than the others in the two cases under consideration, it is always comparable in terms of functionality with the existing solutions, which are both more stable and complete. The main purpose here is to demonstrate the generality of the Polycube programming model and the performance benefits that can be obtained from eBPF-based NFs.

4.8.3 Framework Overheads

In this section we evaluate the overheads imposed by Polycube programming model when compared to baseline eBPF programs written outside the Polycube environment.

4.8.3.1 Overhead for simple NFs

To measure the baseline performance and the overhead introduced by the Polycube abstraction model to a single NF, we implemented the same operations performed by the `xdp_redirect` application [46], available under the Linux samples, as a standalone NF inside the Polycube framework (i.e., `pcn-simplefwd`). The application receives traffic from a given interface and, after swapping the source and destination L2 addresses of the packet, redirects it to a second interface.

Application	Through.	LoC (FP)	LoC (S/CP)
xdp_redirect	6.97Mpps	64	176
tc_redirect	1.60Mpps	53	56
pcn-simplefw (XDP)	6.86Mpps	17	0
pcn-simplefw (TC)	1.55Mpps	17	0

Table 4.3: Comparison between vanilla-eBPF applications and a Polycube network function. All throughput results are single-core.

Table 4.3 shows a comparison between two very simple vanilla eBPF applications and a Polycube NF that performs the same operations, attached to either XDP or Traffic Control (TC) hooks. As we can notice, Polycube introduces a very small overhead compared to vanilla eBPF applications (6.86Mpps vs 6.97Mpps), which is required to provide the abstractions mentioned before (e.g., virtual ports). This is mainly given by the additional processing that happens before and after calling the fast path of the NF, which is totally hidden to the NF developer. As result, the number of LoC for both the fast-path (FP) and the slow and control path (S/CP) is considerably reduced, allowing the developer to focus on the core logic of the program and leaving the common tasks and the possible optimizations to the Polycube daemon. Note also that the sample vanilla applications that we are taking into account are extremely simple; for more complex applications, a developer using vanilla-eBPF has to implement, for example, the entire fast-slow path interaction, which requires a non-negligible amount of effort.

4.8.3.2 Overhead for chain of NFs

Figure 4.12 shows the overhead introduced by the Polycube service chain compared to the standard eBPF tail call mechanism. Of course, eBPF does not support

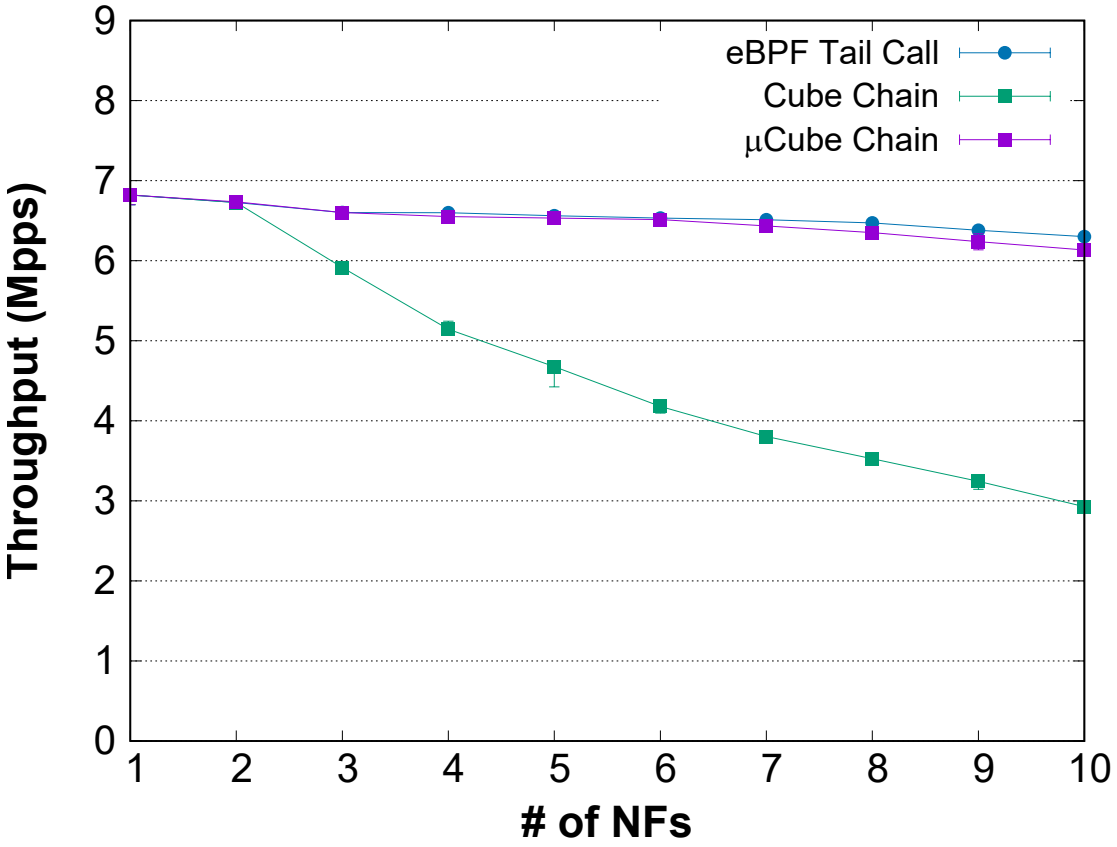


Figure 4.12: Overhead of the Polycube service chain compared to the standard eBPF tail call mechanism.

any type of abstraction required by a NF framework; a tail call performs only an indirect jump from one eBPF program to another. On the other hand, Polycube uses a set of additional components and abstractions that are executed before a packet enters and leaves a NF, e.g., to ensure isolation or virtual port abstraction. This additional overhead is more evident when a packet runs through an increasing number of virtual Cubes but it is necessary to ensure the correct execution of the NF chain. On the other hand, if the same Cube is composed by a set of μCubes the overhead of crossing different μCubes is almost negligible, reflecting the same behavior of the tail call mechanism in the "standard" eBPF approach.

4.8.4 Polycube vs Userspace Frameworks

The difference between kernel and user-space networking is well-known in the literature, with pro and cons on both sides that we partially explored in Section 2. Høiland-Jørgensen et al. [75] have analyzed the performance differences between

XDP and DPDK, showing a gap between the two approaches in favor of the latter; our tests (not reported here for the sake of brevity) simply confirm previous results. This overhead is almost inevitable, and is mainly given by the generality of the Linux operating system, which is structured in a way that make it easier to support different systems and configurations. Edeline et al. [56] have evaluated their userspace VPP [44] based middlebox framework, called `mmb`, against eBPF/XDP-based middlebox implementations, concluding how the latter makes a good in-kernel alternative for their solution. Polycube fills this gap, introducing a negligible overhead over the vanilla-eBPF subsystem (**??**), but greatly simplifying both the development and execution of chains of such in-kernel network services.

4.9 Conclusions

In this Chapter we have presented Polycube, a framework for developing, deploying and managing *in-kernel* virtual network functions. While most of the NFV framework today rely on kernel-bypass approaches, allowing userspace applications to directly access the underlying hardware resources, Polycube brings all the advantages and power of NFV to the work of in-kernel packet processing. It exploits the eBPF subsystem available in the Linux kernel to dynamically inject custom user-defined applications into specific points of the Linux networking stack, providing an unprecedented level of flexibility and customization that would have been unthinkable before. In addition, Polycube has been created with in mind the new requirements brought by the microservice evolution in cloud computing. Polycube services follow the same continuous delivery development of server applications and being able to adapt to continuous configuration and topology changes at runtime, thanks to the possibility to dynamically inject and update existing services without any traffic disruption. At the same time, they offer a level of integration and coexistence with the "traditional" ecosystem that is difficult and inefficient to achieve with kernel-bypass solutions, enabling a high level of introspection and debugging that are fundamental in this environment.

We have implemented a vast range of applications with Polycube and shown that it is not only easy to deploy and to program but also improves network performance of existing in-kernel solutions. Polycube adds a very small overhead compare to vanilla eBPF applications but provides several abstractions that simplify the programming and deployment of new services and enables the creation of complex typologies by concatenating different services together, while maintaining and improving performance.

Chapter 5

Accelerating Linux Security with eBPF iptables

5.1 Introduction

Nowadays, the traditional network security features of a Linux host are centered on `iptables`, which allows applying different security policies to the traffic, such as to protect from possible network threats or to prevent specific communication patterns between different machines. Starting from its introduction in kernel v2.4.0, `iptables` remained the most used packet filtering mechanism in Linux, despite being strongly criticized under many aspects, such as for its far from cutting-edge matching algorithm (i.e., linear search) that limits its scalability in terms of number of policy rules, its syntax, not always intuitive, and its old code base, which is difficult to understand and maintain. In the recent years, the increasing demand of network speed has led to the consciousness that the current implementation may not be able to cope with the modern requirements particularly in terms of performance, flexibility, and scalability [68].

`Nftables` [47] was proposed in 2014 with the aim of replacing `iptables`; it reuses the existing `netfilter` subsystem through an in-kernel virtual machine dedicated to firewall rules, which represents a significant departure from the `iptables` filtering model. Although this yields advantages over its predecessor, `nftables` (and other previous attempts such as `ufw` [161] or `nf-HiPAC` [117]) did not have the desired success, mainly due to the reluctance of the system administrators to adapt their existing configurations (and scripts) operating on the old framework and move into the new one [48]. This is also highlighted by the fact that the majority of today's open-source orchestrators (e.g., Kubernetes [82], Docker [81]) are strongly based on `iptables`.

Recently, another in-kernel virtual machine has been proposed, the extended BPF (eBPF) [8, 151, 65], which offers the possibility to dynamically generate, inject and execute arbitrary code inside the Linux kernel, without the necessity to

install any additional kernel module. eBPF programs can be attached to different hook points in the networking stack such as eXpress DataPath (XDP) [75] or Traffic Control (TC), hence enabling arbitrary processing on the intercepted packets, which can be either dropped or returned (possibly modified) to the stack. Thanks to its flexibility and excellent performance, functionality, and security, recent activities on the Linux networking community have tried to bring the power of eBPF into the newer `nftables` subsystem [14]. Although this would enable `nftables` to converge towards an implementation of its VM entirely based on eBPF, the proposed design does not fully exploit the potential of eBPF, since the programs are directly generated in the kernel and not in userspace, thus losing all the separation and security properties guaranteed by the eBPF code verifier that is executed before the code is injected in the kernel.

On the other hand, `bpfilter` [29] proposes a framework that enables the transparent translation of existing `iptables` rules into eBPF programs; system administrators can continue to use the existing `iptables`-based configuration without even knowing that the filtering is performed with eBPF. To enable such design, `bpfilter` introduces a new type of kernel module that delegates its functionality into user space processes, called *user mode helper* (`umh`), which can implement the rule translation in userspace and then inject the newly created eBPF programs in the kernel. Currently, this work focuses mainly on the design of a translation architecture for `iptables` rules into eBPF instructions, with a small proof of concept that shows the advantages of intercepting (and therefore filtering) the traffic as soon as possible in the kernel, and even in the hardware (smartNICs) [160].

The work presented in this Chapter continues along the `bpfilter` proposal of creating a faster and more scalable clone of `iptables`, but with the following two additional challenges. First is to **preserve the iptables filtering semantic**. Providing a transparent replacement of `iptables`, without users noticing any difference, imposes not only the necessity to respect its syntax but also to implement exactly its behavior; small or subtle differences could create serious security problems for those who use `iptables` to protect their systems. Second is to **improve speed and scalability** of `iptables`; in fact, the linear search algorithm used for matching traffic is the main responsible for its limited scalability particularly in the presence of a large number of firewall rules, which is perceived as a considerable limitation from both the latency and performance perspective.

Starting from the above considerations, this Chapter presents the design of an eBPF-based Linux firewall, called **bpf-iptables**, which implements an alternative filtering architecture in eBPF, while maintaining the same `iptables` filtering semantic but with improved performance and scalability. `bpf-iptables` leverages XDP [75] to provide a fast path for packets that do not need additional processing by the Linux stack (e.g., packets routed by the host) or to discard traffic as soon as it comes to the host. This avoids useless networking stack processing for packets that must be dropped by moving *some* firewall processing off the host CPU entirely,

while still leveraging the rest of the kernel infrastructure to route packets between the different network interfaces.

Our contributions are: *(i)* the design of `bpf-iptables`; it provides an overview of the main challenges and possible solutions in order to preserve the `iptables` filtering semantic given the difference, from hook point perspective, between eBPF and `netfilter`. To the best of our knowledge, `bpf-iptables` is the first application that provides an implementation of the `iptables` filtering in eBPF. *(ii)* A comprehensive analysis of the main limitations and challenges required to implement a fast matching algorithm in eBPF, keeping into account the current limitations chapter 3 of the above technology. *(iii)* A set of data plane optimizations that are possible thanks to the flexibility and dynamic compilation (and injection) features of eBPF, allowing us to create at runtime an optimized data path that fits perfectly with the current ruleset being used.

This Chapter presents the challenges, design choices and implementation of *bpf-iptables* and it compares with existing solutions such as `iptables` and `nftables`. We take into account only the support for the `FILTER` table, while we leave as future work the support for additional features such as `NAT` or `MANGLE`, although they can easily implemented by concatenating different eBPF-based network services that implement such functionalities, as presented in Chapter 4.

5.2 Design Challenges and Assumptions

This Section introduces *(i)* the main challenges encountered while designing `bpf-iptables`, mainly derived from the necessity to emulate the `iptables` behavior with eBPF, and *(ii)* our initial assumptions for this work, which influenced some design decisions.

5.2.1 Guaranteeing filtering semantic

The main difference between `iptables` and `bpf-iptables` lies in their underlying frameworks, `netfilter` and eBPF respectively. `Iptables` defines three default chains for filtering rules associated to the three `netfilter` hooks [135] shown in Figure 5.1, which allow to filter traffic in three different locations of the Linux networking stack. Particularly, those hook points filter traffic that (*i*) *terminates* on the host itself (`INPUT` chain), (*ii*) *traverses* the host such as when it acts as a router and forwards IP traffic between multiple interfaces (the `FORWARD` chain), and (*iii*) *leaves* the host (`OUTPUT` chain).

On the other hand, eBPF programs can also be attached to different hook points. As shown in Figure 5.1, ingress traffic is intercepted in the XDP or traffic control (`TC`) module, hence *earlier* than `netfilter`; the opposite happens for outgoing traffic, which is intercepted *later* than `netfilter`. The different location

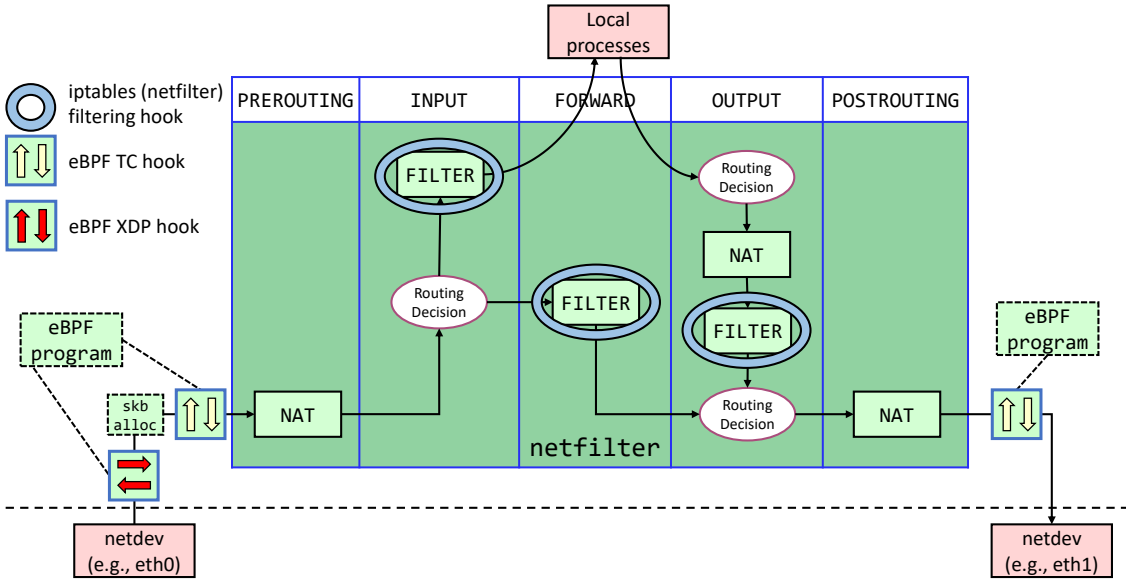


Figure 5.1: Location of `netfilter` and `eBPF` hooks.

of the filtering hooks in the two subsystems introduces the challenge of preserving the semantic of the rules, which, when enforced in an eBPF program, operate on a different set of packets compared to the one that would cross the same `netfilter` chain. For example, rule "`iptables -A INPUT -j DROP`" drops all the incoming traffic crossing the `INPUT` chain, hence directed to the current host; however, it does not affect the traffic forwarded by the host itself, which traverses the `FORWARD` chain. A similar "drop all" rule, applied in the `XDP` or `TC` hook, will instead drop *all* the incoming traffic, including packets that are forwarded by the host itself. As a consequence, `bpf-iptables` must include the capability to predict the `iptables` chain that would be traversed by each packet, maintaining the same semantic although attached to a different hook point.

5.2.2 Efficient classification algorithm in eBPF

The selection and implementation of a better matching algorithm proved to be challenging due to the intrinsic limitations of the eBPF environment chapter 3. In fact, albeit better matching algorithms are well-known in the literature (e.g., cross-producting [148], decision-tree approaches [53, 147, 144, 130, 71, 159]), they require either sophisticated data structures that are not currently available in eBPF[1] or an unpredictable amount of memory, which is not desirable for a module operating at

[1]eBPF programs do not have the right to use traditional memory; instead, they need to rely on a limited set of predefined memory structures (e.g., hash tables, arrays, and a few others), which are used by the kernel to guarantee safety properties and possibly avoid race conditions. As a consequence, algorithms that require different data structures are not feasible in eBPF.

the kernel level. Therefore, the selected matching algorithm must be efficient and scalable, but also feasible with the current eBPF technology.

5.2.3 Support for stateful filters (conntrack)

`Netfilter` tracks the state of TCP/UDP/ICMP connections and stores them in a session (or connection) table (*conntrack*). This table can be used by `iptables` to support stateful rules that accept/drop packets based on the characteristic of the connection they belong to. For instance, `iptables` may accept only outgoing packets belonging to `NEW` or `ESTABLISHED` connections, e.g., enabling the host to generate traffic toward the Internet (and to receive return packets), while connections initiated from the outside world may be forbidden. As shown in Figure 5.1, `bpf-iptables` operates *before* packets enter in `netfilter`; being unable to exploit the Linux *conntrack* module to classify the traffic, it has to implement its own equivalent component (Section 5.4.5.)

5.2.4 Working with upstream Linux kernel

Our initial assumption for this work is to operate with the existing Linux kernel in order to bring the advantages of `bpf-iptables` to the wider audience as soon as possible. In fact, the process required by the Linux kernel community to agree with any non-trivial code change and have them available in a mainline kernel is rather long and may easily require more than one year. This assumption influenced, in some cases, the design choices taken within `bpf-iptables` (e.g., the definition of a new conntrack in eBPF instead of relying on the existing Linux one); we further analyze this point in Section 5.7, providing a discussion of the possible modification to the eBPF subsystem that could further improve `bpf-iptables`.

5.3 Overall Architecture

Figure 5.2 shows the overall system architecture of `bpf-iptables`. The data plane includes four main classes of eBPF programs. The first set (blue) implements the classification pipeline, i.e., the ingress, forward or output chain; a second set (yellow) implements the logic required to preserve the semantics of `iptables`; a third set (orange) is dedicated to connection tracking. Additional programs (grey) are devoted to ancillary tasks such as packet parsing.

The *ingress* pipeline is called upon receiving a packet either on the XDP or TC hook. By default, `bpf-iptables` works in XDP mode, attaching all the eBPF programs to the XDP hook of the host's interfaces. However, this requires the

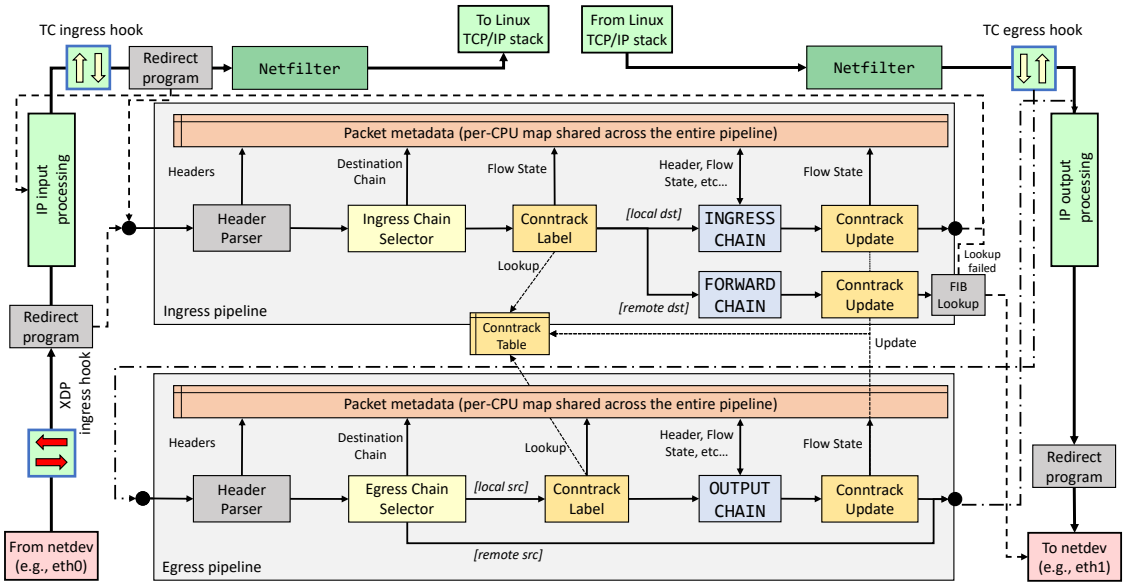


Figure 5.2: High-level architecture of `bpf-iptables`.

explicit support for XDP in the NIC drivers[2]; `bpf-iptables` automatically falls back to the TC mode when the NIC drivers are not XDP-compatible. In the latter case, all the eBPF programs composing the *ingress* pipeline are attached to the TC hook. The *egress* pipeline is instead called upon receiving a packet on the TC egress hook, before the packet leaves the host, as XDP is not available in egress [39].

Once in the *ingress* pipeline, the packet can enter either the `INPUT` or `FORWARD` chain depending on the routing decision; in the first case, if the packet is not dropped, it will continue its journey through the Linux TCP/IP stack, ending up in a local application. In the second case, if the `FORWARD` pipeline ends with an `ACCEPT` decision, `bpf-iptables` redirects the packet to the target NIC, without returning it to the Linux networking stack (more details in Section 5.4.4.2). On the other hand, a packet leaving the host triggers the execution of `bpf-iptables` when it reaches the TC egress hook, where it will be processed by the `OUTPUT` chain.

Finally, a control plane module (not depicted in Figure 5.2) is executed in userspace and provides three main functions: *(i)* initialization and update of the `bpf-iptables` data plane, *(ii)* configuration of the eBPF data structures required to run the classification algorithm and *(iii)* monitoring for changes in the number and state of available NICs, which is required to fully emulate the behavior of `iptables`, handling the traffic coming from *all* the host interfaces. We will describe the design and architecture of the `bpf-iptables` data plane in Section 5.4, while the operations performed by the control plane will be presented in Section 5.5.

[2]NIC driver with native support for XDP can be found at [36].

5.4 Data plane

In the following subsections we present the different components belonging to the `bpf-iptables` data plane, as shown in Figure 5.2.

5.4.1 Header Parser

The `bpf-iptables` ingress and egress pipelines start with a *Header Parser* module that extracts the packet headers required by the current filtering rules, and stores each field value in a per-CPU array map shared among all the eBPF programs in the pipeline, called *packet metadata*. This avoids the necessity of packet parsing capabilities in the subsequent eBPF programs and guarantees both better performance and a more compact processing code. The code of the *Header Parser* is dynamically generated on the fly; when a new filtering rule that requires the parsing of an additional protocol field is added, the control plane re-generates, compiles and re-injects the obtained eBPF program in the kernel in order to extract also the required field. As a consequence, the processing cost of this block is limited exactly to the number of fields that are currently needed by the current `bpf-iptables` rules.

5.4.2 Chain Selector

The *Chain Selector* is the second module in the data plane whose role is to classify and forward the traffic to the correct classification pipeline (i.e., chain), according to the `iptables` semantic (Section 5.2.1). It anticipates the routing decision that would have been performed later in the TCP/IP stack and predicts the right chain that will be hit by the current packet. The traffic coming from a network interface would cross the `INPUT` chain only if it is directed to a *local* IP address, visible from the host root namespace, while incoming packets directed to a *non-local* IP address would cross the `FORWARD` chain. On the other hand, an outgoing packet traverses the `OUTPUT` chain only if it has been generated locally, i.e., by a *local* IP address. To achieve this behavior, `bpf-iptables` uses a separate Chain Selector module for the ingress and egress pipeline.

The Ingress Chain Selector checks if the *destination* IP address of the incoming packet is present in the `BPF_HASH` map that keeps local IPs and writes the resulting target chain in the *packet metadata* per-CPU map shared across the entire pipeline. This value is used by the next module of the chain (i.e., the conntrack) to jump to the correct target chain. On the other hand, the Egress Chain Selector, which is part of the egress pipeline, classifies traffic based on the *source* IP address and sends it to either the `OUTPUT` chain or directly to the output interface. In fact, traffic traversing the `FORWARD` chain has already been matched in the ingress pipeline, hence it should not be handled by the `OUTPUT` chain.

5.4.3 Matching algorithm

To get over the linear search performance of `iptables`, `bpf-iptables` adopts the more efficient Linear Bit-Vector Search (LBVS) [98] classification algorithm. LBVS provides a reasonable compromise between feasibility and speed; it has an intrinsic pipelined structure which maps nicely with the eBPF technology, hence enabling the optimizations presented in Section 5.4.4.2. The algorithm follows the *divide-and-conquer* paradigm: it splits filtering rules in multiple classification steps, based on the number of protocol fields in the ruleset; intermediate results that carry the potentially matching rules are combined to obtain the final solution.

Classification. LBVS requires a specific (logical) bi-dimensional table for each field on which packets may match, such as the three fields shown in the example of Figure 5.3. Each table contains the list of unique values for that field present in the given ruleset, plus a wildcard for rules that do not care for any specific value. Each value in the table is associated with a bitvector of length N equal to the number of rules, in which the i^{th} '1' bit tells that rule i may be matched when the field assumes that value. Filtering rules, and the corresponding bits in the above bitvector, are ordered with highest priority rule first. The matching process is repeated for each field we operate with, such as the three fields shown in Figure 5.3. The final matching rule can be obtained by performing a bitwise `AND` operation on all the intermediate bitvectors returned in the previous steps and determining the most significant '1' bit in the resulting bitvector. This represents the matched rule with the highest priority, which corresponds to rule #1 in the example in Figure 5.3. Bitmaps enable the evaluation of rules in large batches, which depend on the parallelism of the main memory; while still theoretically a linear algorithm, this scaling factor enables a 64x speedup compared to a traditional linear search on common CPUs.

5.4.4 Classification Pipeline

The `bpf-iptables` classification pipeline (Figure 5.4) is in charge of filtering packets according to the rules configured for a given chain. It is made by a sequence of eBPF programs, each one handling a single matching protocol field of the current ruleset. The pipeline contains two per-CPU shared maps that keep some common information among all the programs, such as the temporary bitvector containing the partial matching result, which is initialized with all the bits set to '1' before a packet enters the pipeline.

Each module of the pipeline performs the following operations: *(i)* extracts the needed packet fields from the *packet metadata* map, previously filled by the *Header Parser* module; *(ii)* performs a lookup on its private eBPF map to find the bitvector

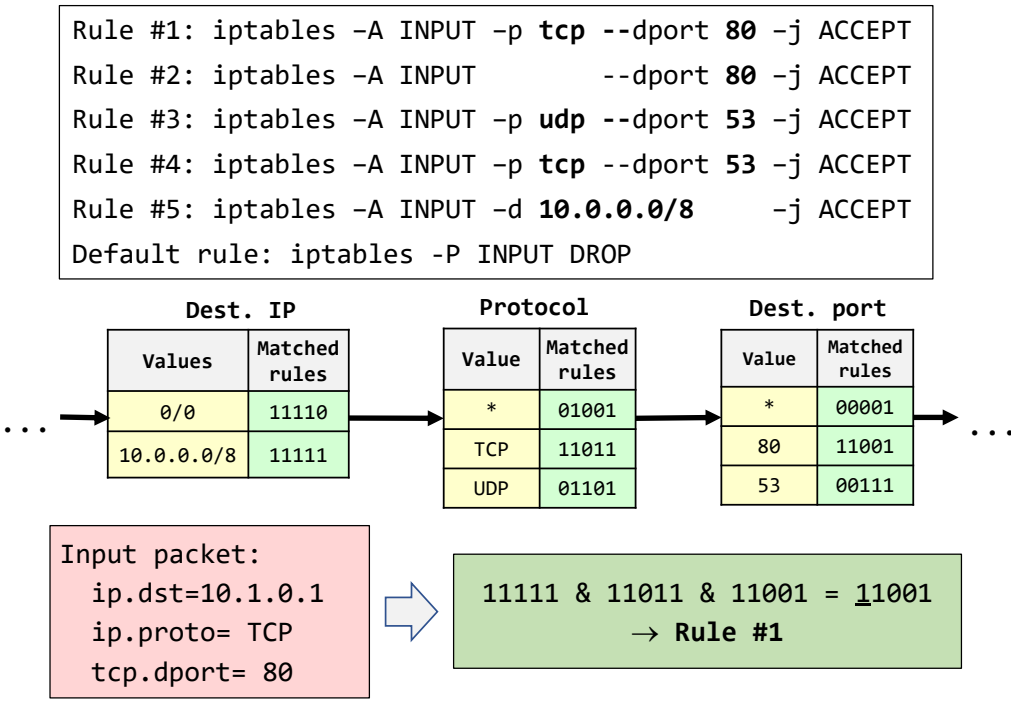


Figure 5.3: Linear Bit Vector Search

associated to the current packet value for that field. If the lookup succeeds, *(iii-a)* it performs a `bitwise AND` between this bitvector and the *temporary* bitvector contained in the per-CPU map. If the lookup fails and there is a wildcard rule, *(iii-b)* the `AND` is performed between the bitvector associated with the wildcard rules and the one present in the per-CPU map. Instead, *(iii-c)* if the lookup fails and there are no wildcard rules for that field, we can immediately conclude that the current packet does not match any rule within the ruleset; hence, we can exploit this situation for an early break of the pipeline (Section 5.4.4.2). Finally, except the last case, *(iv)* it saves the new bitvector in the shared map and calls the next module of the chain.

Bitvectors comparison. Since each matching rule is represented as a '1' in the bitvector, `bpf-iptables` uses an array of N 64bit unsigned integers to support a large number of rules (e.g., 2.048 rules can be represented as an array of 32 `uint64_t`). As consequence, when performing the `bitwise AND`, the current eBPF program has to perform N cycles on the entire array to compare the two bitvectors. Given the lack of loops on eBPF, this process requires loop unrolling and is therefore limited by the maximum number of possible instructions within an eBPF program, thus also limiting the maximum number of supported rules. The necessity to perform loop unrolling is, as consequence, the most compelling reason for

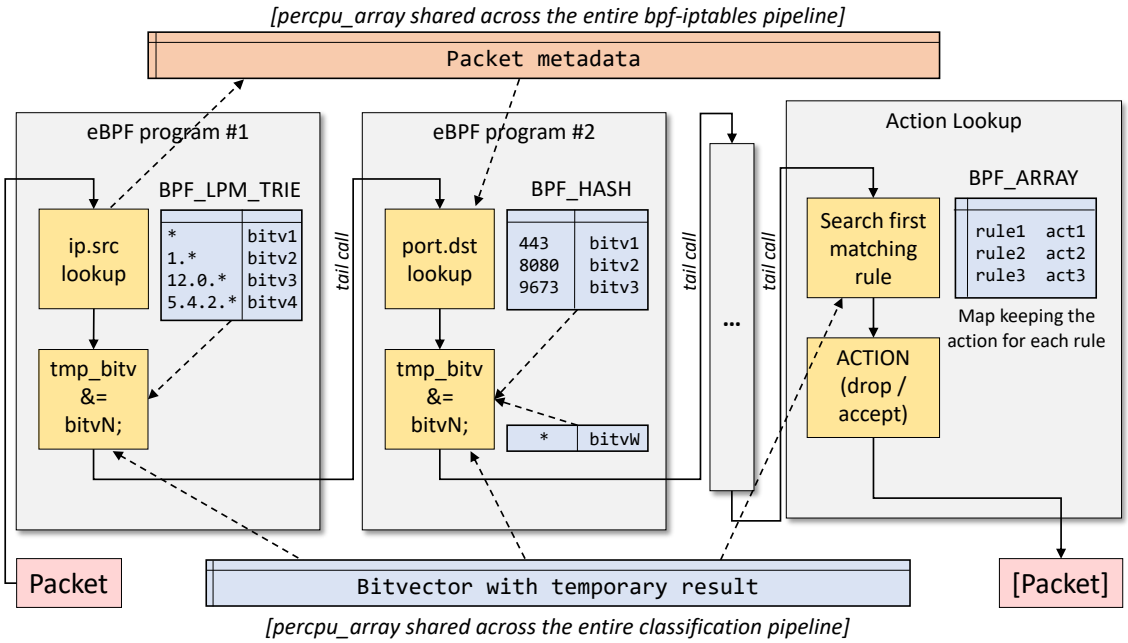


Figure 5.4: `bpf-iptables` classification pipeline.

splitting the classification pipeline of `bpf-iptables` across many eBPF modules, instead of concentrating all the processing logic within the same eBPF program.

Action lookup. Once we reach the end of the pipeline, the last program has to find the rule that matched the current packet. This program extracts the bitvector from the per-CPU shared map and looks for the position of the first bit to 1 in the bitvector, using the de Bruijn sequences [102] to find the index of the first bit set in a single word; once obtained, it uses that position to retrieve the final action associated with that rule from a given `BPF_ARRAY` map and finally applies the action. Obviously, if no rules have been matched, the default action is applied.

5.4.4.1 Clever data sharing

`bpf-iptables` makes a massive use of eBPF per-CPU maps, which represent memory that can be shared among different cascading programs, but that exist in multiple independent instances equal to the number of available CPU cores. This memory structure guarantees very fast access to data, as it statically assigns a set of memory locations to each CPU core; consequently, data is never realigned with other L1 caches present on other CPU cores, hence avoiding the (hidden) hardware cost of cache synchronization. Per-CPU maps represent the perfect choice in our scenario, in which multiple packets can be processed in parallel on different CPU

cores, but where all the eBPF programs that are part of the same chain are guaranteed to be executed on the same CPU core. As a consequence, all the programs processing a packet P are guaranteed to have access to the same shared data, without performance penalties due to possible cache pollution, while multiple processing pipelines, each one operating on a different packet, can be executed in parallel. The consistency of data in the shared map is guaranteed by the fact that eBPF programs are never preempted by the kernel (even across tail calls). They can use the per-CPU map as a sort of stack for temporary data, which can be subsequently obtained from the downstream program in the chain with the guarantees that data are not overwritten during the parallel execution of another eBPF program on another CPU and thus ensuring the correctness of the processing pipeline.

5.4.4.2 Pipeline optimizations

Thanks to the modular structure of the pipeline and the possibility to regenerate part of it at runtime, we can adopt several optimizations that allow *(i)* to jump out of the pipeline when we realize that the current packet does not require further processing and *(ii)* to modify and rearrange the pipeline at runtime based on the current `bpf-iptables` ruleset values.

Early-break. While processing a packet, `bpf-iptables` can discover in advance that it will not match any rule. This can happen in two separate cases. The first occurs when, at any step of the pipeline, a lookup in the bitvector map fails; in such event, if that field does not have a wildcard value, we can directly conclude that the current packet will not match any rule. The second case takes place when the result of the bitwise AND between the two bitvectors is the empty set (all bits set to 0). In either circumstance, the module that detects this situation can jump out of the pipeline by applying the default policy for the chain, without the additional overhead of executing all the following components. If the policy is `DROP`, the packet is immediately discarded concluding the pipeline processing; if the default policy is `ACCEPT`, the packet will be delivered to the destination, before being processed by the *Conntrack Update* module (Section 5.4.5).

Accept all established connections. A common configuration applied in most `iptables` rulesets contains an `ACCEPT` all `ESTABLISHED` connections as the first rule of the ruleset. When the `bpf-iptables` control plane discovers this configuration in a chain, it forces the *Conntrack Label* program to skip the classification pipeline if it recognizes that a packet belongs to an `ESTABLISHED` connection. Since this optimization is performed per-chain (we could have different configurations among the chains), the Conntrack Label module reads the target chain from the *packet metadata* per-CPU map previously filled by the Chain Selector and immediately performs a *tail-call* to the final connection tracking module that will update the conntrack table accordingly (e.g., updating the timestamp for that connection).

Optimized pipeline. Every time the current ruleset is modified, `bpf-iptables` creates a processing pipeline that contains the minimum (optimal) number of processing blocks required to handle the fields of the current ruleset, avoiding unnecessary processing. For instance, if there are no rules matching TCP flags, that processing block is not added to the pipeline. New processing blocks can be dynamically added at run-time if the matching against a new field is required. In addition, `bpf-iptables` is able to re-organize the classification pipeline by changing the order of execution of the various components. For example, if some components require only an exact matching, a match failed on that field would lead to an early-break of the pipeline; putting those modules at the beginning of the pipeline could speed up processing, avoiding unnecessary memory accesses and modules.

HOmogeneous RUleset analySis (HORUS). This optimization (that we called `HORUS`) is used to (partially) overcome two main restrictions of `bpf-iptables`: the maximum number of matching rules, given by the necessity to perform loop unrolling to compare the bitvectors, and the rule updating time, since we need to re-compute all the bitvectors when the ruleset is updated. The idea behind HORUS is based on the consideration that often, firewall rulesets (in particular, the ones automatically configured by orchestration software), contain a set of *homogeneous* rules that operate on the same set of fields. If we are able to discover this set of "similar" rules that are not conflicting with the previous ones (with higher priority), we could bring them in front of the matching pipeline for an additional chance of early-break. In addition, since those rules are independent from the others in the ruleset, we could compact all their corresponding bits in the bitvectors with just one, hence increasing the space for other non-`HORUS` rules. Moreover, adding (or removing) a `HORUS` rule does not require to update or even change the entire matching pipeline, but a single map insertion (or deletion) would be enough, thus reducing considerably the rule update time. When enabled, the `HORUS` module is inserted right before the Conntrack Label on and consists of another eBPF program with a `BPF_HASH` table that contains, as key, the set of fields of the `HORUS` set and, as value, the final action to apply when a match is found. If the final action is `DROP`, the packet is immediately dropped; if the action is `ACCEPT`, it will directly jump to the last module of the pipeline, the Conntrack Update. Finally, if no match is found, `HORUS` jumps to the first program of the classification pipeline, following the usual processing path.

An important scenario where `HORUS` shows its great advantages is under **DoS attacks**. In fact, if all the rules of the `HORUS` ruleset contains a `DROP` action, matching packets will be immediately discarded, hence exploiting *(i)* the early processing provided by XDP that allows to drop packets at a high rate and *(ii)* the ability to run this program on hardware accelerators (e.g., SmartNICs) that support the offloading of "simple" eBPF programs, further reducing the system load and the resource consumption, as shown in Chapter 6.

Optimized forwarding. If the final decision for a packet traversing the `FORWARD` chain is `ACCEPT`, it has to be forwarded to the next-hop, according to the routing table of the host. Since, starting from kernel version 4.18, eBPF programs can query directly the Linux routing table, `bpf-iptables` can optimize the path of the above packet by directly forwarding the packet to the target NIC, shortening its route within the Linux stack, with a significant performance advantage (Section 5.6). In the (few) cases in which the needed information are not available (e.g., because the MAC address of the next hop is not yet known), `bpf-iptables` will deliver the first few packets to the Linux stack, following the usual path.

5.4.4.3 Atomic rule update

One of the characteristics of the LBVS classifier is that, whenever a new rule is added, updated or removed, it needs to re-compute all the bitvectors associated with the current fields. However, to avoid inconsistency problems, we must update *atomically* the content of *all* maps in the pipeline. Unfortunately, eBPF allows the atomic update of a *single* map, while it does not support atomic updates of multiple maps. Furthermore, defining a synchronization mechanism for the update (e.g., using locks to prevent traffic being filtered by `bpf-iptables`) could lead to unacceptable service disruption given the impossibility of the data plane to process the traffic in that time interval.

To solve this issue, `bpf-iptables` exploits the fact that the classification pipeline is stateless and therefore it creates a new chain of eBPF programs and maps in parallel, based on the new ruleset. While this new pipeline is assembled and injected in the kernel, packets continue to be processed in the initial matching pipeline, accessing to the current state and configuration; when this reloading phase is completed, the Chain Selector is updated to jump to the first program of the new chain, allowing new packets to flow through it. This operation is performed *atomically*, enabling the continuous processing of the traffic with a consistent state and without any service disruption, thanks to a property of the eBPF subsystem that uses a particular map (`BPF_PROG_ARRAY`) to keep the addresses of the instantiated eBPF programs. Finally, when the new chain is up and running, the old one is unloaded. We discuss and evaluate the performance of the rules update within `bpf-iptables`, `iptables` and `nftables` in Section 5.6.4.2.

5.4.5 Connection Tracking

To support stateful filters, `bpf-iptables` implements its own connection tracking module, which is characterized by four additional eBPF programs placed in both ingress and egress pipeline, plus an additional matching component in the classification pipeline that filters traffic based on the current connection's state. These modules share the same `BPF_HASH` *conntrack* map, as shown in Figure 5.2.

To properly update the state of a connection, the `bpf-iptables` conntrack has to intercept the traffic in both directions (i.e., host to the Internet and vice versa). Even if the user installs a set of rules operating only on the `INPUT` chain, outgoing packets have to be processed, in any case, by the conntrack modules located in the egress pipeline. The `bpf-iptables` connection tracking supports TCP, UDP, and ICMP traffic, although it does not handle advanced features such as *related* connections (e.g., when a SIP control session triggers the establishment of voice/video RTP flows[3]), nor it supports IP reassembly.

Packet walkthrough. The *Conntrack Label* module is used to associate a label to the current packet[4] by detecting any possible change in the *conntrack* table (e.g., `TCP SYN` packet starting a new connection triggers the creation of a new session entry), which is written into the *packet metadata* per-CPU map shared within the entire pipeline. This information is used to filter the packet according to the stateful rules of the ruleset. Finally, if the packet "survives" the classification pipeline, the second conntrack program (*Conntrack Update*) updates the *conntrack* table with the new connection state or, in the case of a new connection, it creates the new associated entry. Since no changes occur if the packet is dropped, forbidden sessions will never consume space in the connection tracking table.

Conntrack entry creation. To identify the connection associated to a packet, `bpf-iptables` uses the packet 5-tuple (i.e., src/dst IP address, L4 protocol, src/dst L4 port) as key in the *conntrack* table. Before saving the entry in the table, the *Conntrack Update* orders the key as follows:

$$\begin{aligned} key = \{&min(IpSrc, IpDest), max(IpSrc, IpDest), Proto, \\ &min(PortSrc.PortDest), max(PortSrc, PortDest)\} \end{aligned} \tag{5.1}$$

This process allows to create a single entry in the *conntrack* table for both directions, speeding up the lookup process. In addition, together with the new connection state, the *Conntrack Update* module stores into the *conntrack* table two additional flags, *ip reverse* (`ipRev`) and *port reverse* (`portRev`) indicating if the IPs and the L4 ports have been reversed compared to the current packet 5-tuple. Those information will be used during the lookup process to understand if the current packet is in the same direction as the one originating the connection, or the opposite.

[3]eBPF programs can read the payload of the packet (e.g., [7]), which is required to recognize *related* connections. Supporting these features in `bpf-iptables` can be done by extending the conntrack module to recognize the different L7 protocol from the packet and inserting the correct information in the conntrack table.

[4]The possible labels that the conntrack module associates to a packet are the same defined by the `netfilter` framework (i.e., `NEW, ESTABLISHED, RELATED, INVALID`).

Lookup process. When a packet arrives to the *Conntrack Label* module, it computes the key for the current packet according to the previous formula and determines the *ip reverse* and *port reverse* flags as before. At this point it performs a lookup into the *conntrack* table with this key; if the lookup succeeds, the new flags are compared with those saved in the *conntrack* table to detect which direction the packet belongs to. For instance, if:

```
(currIpRev != IpRev) && (currPortRev != PortRev)
```
(5.2)

we are dealing with the reverse packet related to the stored session; this is used, e.g., to mark an existing TCP session as ESTABLISHED, i.e., update its state from SYN_SENT to SYN_RCVD (Figure 5.5).

Stateful matching module. If at least one rule of the ruleset requires a stateful match, `bpf-iptables` instantiates also the *Conntrack Match* module within the classification pipeline to find the bitvector associated to the current label. While this module is present only when the ruleset contains stateful rules, the two connection tracking modules outside the classification pipeline are always present, as they have to track all the current connections in order to be ready for state-based rules instantiated at a later time.

TCP state machine. A summary of the TCP state machine implemented in the connection tracking module is shown in Figure 5.5. The first state transition is triggered by a `TCP SYN` packet (all other packets not matching that condition are marked with the `INVALID` label); in this case, if the packet is accepted by the classification pipeline, the new state (i.e., `SYN_SENT`) is stored into the *conntrack* table together with some additional flow context information such as the last seen sequence number, which is used to check the packet before updating the connection state. Figure 5.5 refers to *forward* or *reverse* packet (i.e., `pkt` or `rPkt`) depending on the initiator of the connection. Finally, when the connection reaches the `TIME_WAIT` state, only a timeout event or a new `SYN` will trigger a state change. In the first case the entry is deleted from the *conntrack* table, otherwise the current packet direction is marked as forward and the new state becomes `SYN_SENT`.

Conntrack Concurrency. Even though the conntrack table, as all the others eBPF maps, is protected by the kernel's RCU mechanism, it is not enough to ensure the atomicity of the operations performed inside the conntrack module itself. Indeed, to correctly work, the algorithm presented above assumes that both the *direct* and *reverse* connection are received on the same CPU core, requiring RSS/RPS[5]

[5]The Receive Side Scaling (RSS) is a hardware mechanism implemented in the NIC itself to distribute different flows to multiple receive and transmit descriptor queues, which are generally mapped to different CPU cores. The Receive Packet Steering (RPS) is logically a software

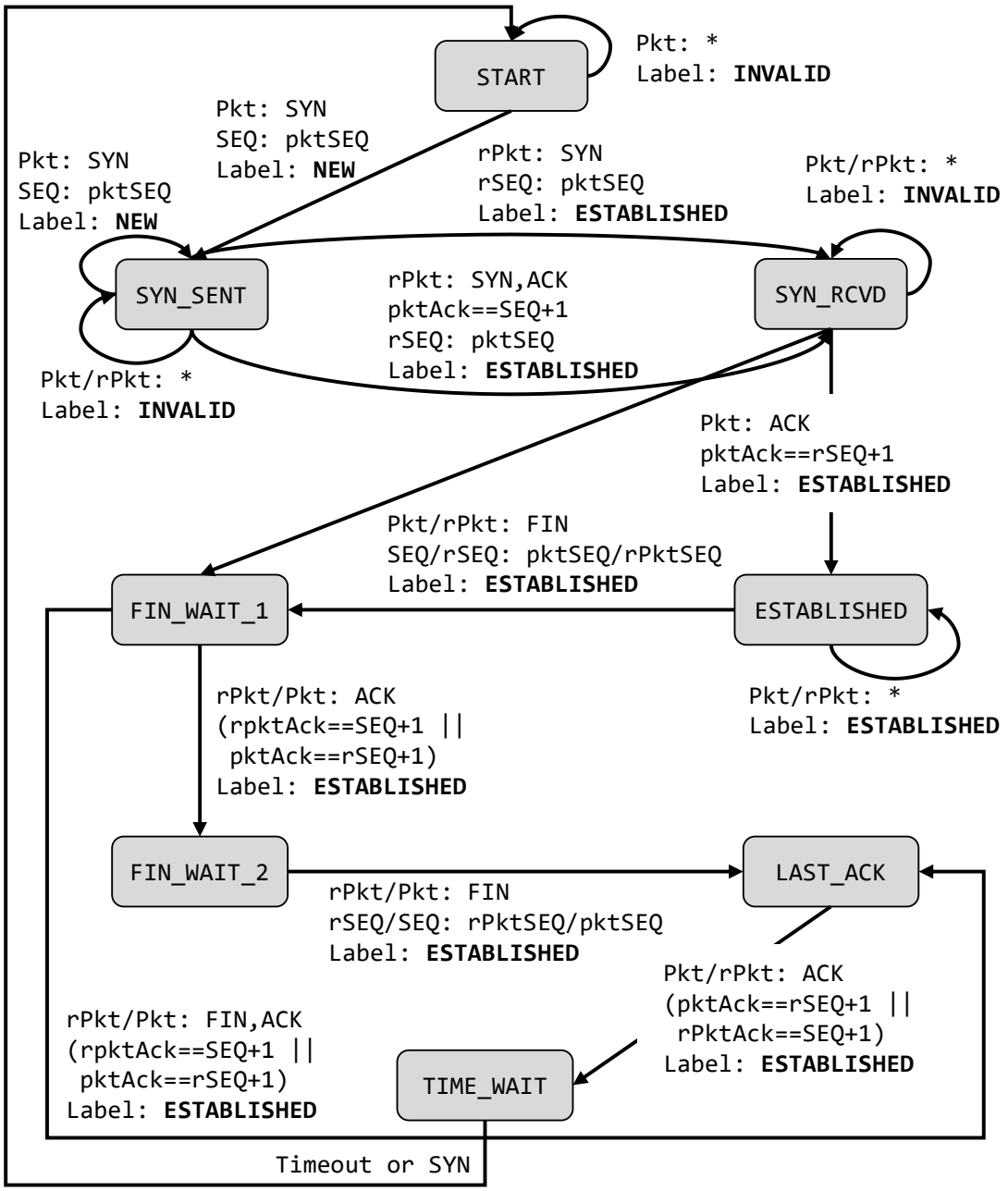


Figure 5.5: TCP state machine for `bpf-iptables` conntrack. Grey boxes indicate the states saved in the conntrack table; labels represent the value assigned by the first conntrack module before the packet enters the classification pipeline.

implementation of RSS and it is used when RSS is not supported by the underlying NIC.

to be enabled on the system. However, this assumption alone it is still not enough to ensure the correctness the entire conntrack module. In fact, before updating the value, the *Conntrack Update* first checks the current state of the packet in the state machine and then updates the value with the new state. Although the update can happen atomically through the `bpf_map_update()` system call, it is not the same for both operations (check and update), then causing inconsistencies between the two conntrack modules. `bpf-iptables` solves this problem by using (for v5.1+ kernels) the `bpf_spin_lock` mechanism [49, 152], which allows to acquire a lock on a specific conntrack entry and then perform, atomically, the read and update of the value.

Conntrack Cleanup. `bpf-iptables` implements the cleanup of conntrack entries in the control plane, where a dedicated thread checks the presence of expired sessions. For this reason, the *Conntrack Update* module updates the *timestamp* associated to session each entry when a new packet is received. Since we noticed that the usage of the `bpf_ktime()` helper to retrieve the current timestamp causes a non-negligible performance overhead, we store in a dedicated per-CPU array the current time every *second*, which is used by the data plane to timestamp the session entries. We are confident that a per-second precision is a reasonable trade-off between performance and accuracy for this type of application.

5.5 Control plane

This Section describes the main operations of our control plane, which are triggered whenever one of the following events occur.

Start-up. To behave as `iptables`, `bpf-iptables` has to intercept all incoming and outgoing traffic and handle it in its custom eBPF pipeline. When started, `bpf-iptables` attaches a small eBPF *redirect* program to the ingress (and egress) hook of each host's interface visible from the root namespace, as shown in Figure 5.2. This program intercepts all packets flowing through the interface and calls the first program of the `bpf-iptables` ingress or egress pipeline. This enables the creation of a single processing pipeline that handles all the packets, whatever interface they come from, as eBPF programs attached to a NIC cannot be called from other interfaces. Finally, `bpf-iptables` retrieves all local IP addresses active on any NIC and configures them in the *Chain Selector*; this initialization phase is done by subscribing to the proper set of `netlink` events.

Netlink notification. When a new `netlink` notification arrives, `bpf-iptables` checks if it relates to specific events in the root namespace, such as the creation of an interface or the update of an IP address. In the first case, the *redirect* program

Algorithm 1: Pre-processing algorithm

Input: N, the list of filtering rules

```
1  Extract K, the set of matching fields used in N
2  foreach k_i ∈ K do
3      b_i ← # bit of field K_i
4      θ_i ← {k_{i,j} | ∀j ≤ min (card(N), 2^{b_i})}  /* set of distinct values */
5      if ∃ a wildcard rule ∈ N for k_i then
6          Add wildcard entry to θ_i
7      foreach k_{i,j} ∈ θ_i do
8          bitvector_{i,j}[N] ← {0}
9          foreach n_i ∈ N do
10             if k_{i,j} ⊆ n_i then
11                 bitvector_{i,j}[i] = 1
```

is attached to the eBPF hook of the new interface, enabling `bpf-iptables`[6] to inspect its traffic. In the second case, we update the list of local IPs used in the *Chain Selector* with the new address.

Ruleset changes. When the user updates the ruleset, `bpf-iptables` starts the execution of the **pre-processing** algorithm, which calculates the value-bitvector pairs for each field; those values are then inserted in the new eBPF maps and the new programs are created on the parallel chain. The pre-processing algorithm (pseudo-code in Algorithm 1) works as follows. Let's assume we have a list of N packet filtering rules that require exact or wildcard matching on a set of K fields; *(i)* for each field $k_i \in K$ we extract a set of distinct values $\theta_i = \{k_{i,1}, k_{i,2}, ..., k_{i,j}\}$ with $j \leq card(N)$ from the current ruleset N; *(ii)* if there are rules that require wildcard matching for the field k_i, we add an additional entry to the set θ_i that represents the wildcard value; *(iii)* for each $k_{i,j} \in \theta_i$ we scan the entire ruleset and if $\forall n_i \in N$ we have that $k_{i,j} \subseteq n_i$ then we set the bit corresponding to the position of the rule n_i in the bitvector for the value $k_{i,j}$ to 1, otherwise we set the corresponding bit to 0. Repeating these steps for each field $k_i \in K$ will allow to construct the final value-bitvector pairs to be used in the classification pipeline.

The final step for this phase is to insert the generated values in their eBPF maps. Each matching field has a default map; however, `bpf-iptables` is also able

[6]There is a transition window between the reception of the `netlink` notification and the load of the *redirect* program, during which the firewall is not yet active. As far as the eBPF is concerned, this transition cannot be totally removed.

to choose the map type at runtime, based on the current ruleset values. For example, a `LPM_TRIE` is used as default map for IP addresses, which is the ideal choice when a range of IP addresses is used; however, if the current ruleset contains only rules with fixed (/32) IP addresses, it changes the map into a `HASH_TABLE`, making the matching more efficient. Before instantiating the pipeline, `bpf-iptables` modifies the behavior of every single module by regenerating and recompiling the eBPF program that best represents the current ruleset. When the most appropriate map for a given field has been chosen, `bpf-iptables` fills it with computed value-bitvector pairs. The combination of eBPF map and field type affects the way in which `bpf-iptables` represents the wildcard rule. For maps such as the `LPM_TRIE`, used to match IP addresses, the wildcard can be represented as the value 0.0.0.0/0, which is inserted as any other value. On the other hand, for L4 source and destination ports, which use a `HASH_MAP`, `bpf-iptables` instantiates the wildcard value as a variable hard-coded in the eBPF program; when the match in the table fails, it will use the wildcard variable as it was directly retrieved from the map.

`Bpf-iptables` adopts a variant of the previous algorithm for fields that have a limited number of possible values, where instead of generating the set θ_i of distinct values for the field k_i, it produces all possible combinations for that value. The advantage is that *(i)* it does not need to generate a separate bitvector for the wildcard, being all possible combinations already contained within the map and *(ii)* can be implemented with an eBPF `ARRAY_MAP`, which is faster compared to other maps. An example is the processing of TCP flags; since the number of all possible values for this field is limited (i.e., 2^8), it is more efficient to expand the entire field with all possible cases instead of computing exactly the values in use.

5.6 Evaluation

5.6.1 Test environment

Setup. Our testbed includes a first server used as DUT running the firewall under test and a second used as packet generator (and possibly receiver). The DUT encompasses an Intel Xeon Gold 5120 14-cores CPU @2.20GHz (hyper-threading disabled) with support for Intel's Data Direct I/O (DDIO) [84], 19.25 MB of L3 cache and two 32GB RAM modules. The packet generator is equipped with an Intel® Xeon CPU E3-1245 v5 4-cores CPU @3.50GHz (8 cores with hyper-threading), 8MB of L3 cache and two 16GB RAM modules. Both servers run Ubuntu 18.04.1 LTS, with the packet generator using kernel 4.15.0-36 and the DUT running kernel 4.19.0. Each server has a dual-port Intel XL710 40Gbps NIC, each port directly connected to the corresponding one of the other server.

Evaluation metrics. Our tests analyze both TCP and UDP throughput of `bpf-iptables` compared to existing (and commonly used) Linux tools, namely `iptables` and `nftables`. TCP tests evaluate the throughput of the system under "real" conditions, with all the offloading features commonly enabled in production environments (i.e., IP fragmentation, TCP segmentation offloading, checksum offloading). Instead, UDP tests stress the capability of the system in terms of packet per seconds, hence we use 64B packets without any offloading capability. When testing `bpf-iptables`, we disabled all the kernel modules related to `iptables` and `nftables` (e.g., `x_tables`, `nf_tables`) and the corresponding connection tracking modules (i.e., `nf_conntrack` and `nft_ct`). Although most of the evaluation metrics are common among all tests, we provide additional details on how the evaluation has been performed on each test separately.

Testing tools. UDP tests used `Pktgen-DPDK` v3.5.6 [55] and `DPDK` v18.08 to generate traffic, while for TCP tests we used both `iperf` v2.0.10 to measure the TCP throughput and `weighttp` [10] v0.4 to generate a high number of *new* parallel HTTP connection towards the DUT, counting only the successful completed connections [91]. Particularly, the latter reports the actual capability of the server to perform real work.

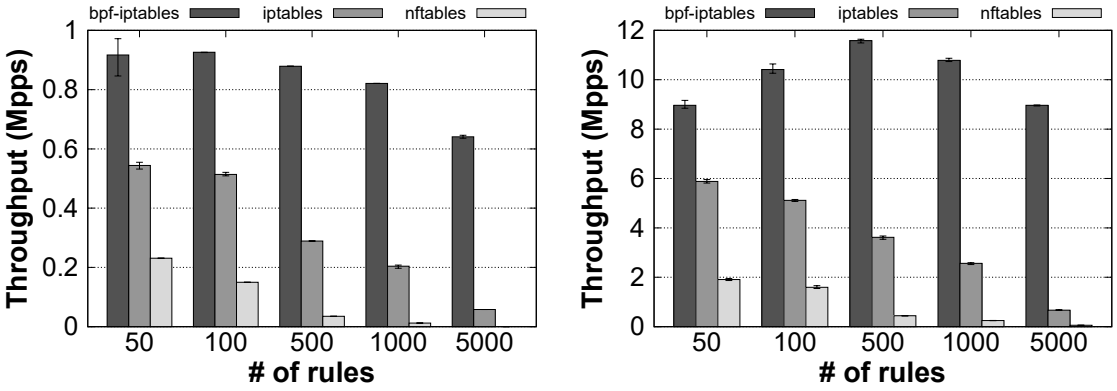
Rulesets and Packet-traces. We used the same ruleset for all the firewalls under consideration. In particular, `nftables` rules have been generated using the same rules loaded for `bpf-iptables` and `iptables` but converted using *iptables-translate* [11]. Since synthetic rulesets vary depending on the test under consideration, we describe their content in the corresponding test's section. Regarding the generated traffic, we configured `Pktgen-DPDK` to generate traffic that matches the configured rules; also in this case we discuss the details in each test description.

5.6.2 System benchmarking

This Section evaluates the performance and efficiency of individual `bpf-iptables` components (e.g., conntrack, matching pipeline).

5.6.2.1 Performance dependency on the number of rules

This test evaluates the performance of `bpf-iptables` with an increasing number of rules, from 50 to 5k. We generated five synthetic rulesets with rules matching the TCP/IP 5-tuple and then analyzed a first scenario in which rules are loaded on the `FORWARD` chain (Section 5.6.2.2) and a second that involves the `INPUT` chain (Section 5.6.2.3). In the first case, performance are influenced by both the classification algorithm and the TCP/IP stack bypass; in the second case packets are



(a) (b)

Figure 5.6: Single 5.6a and multi-core 5.6b comparison when increasing the number of loaded rules. Generated traffic (64B UDP packets) is uniformly distributed among all the rules.

delivered to a local application, hence the performance are mainly influenced by the classification algorithm.

5.6.2.2 Performance dependency on the number of rules (`FORWARD` chain)

This test loads all the rules in the `FORWARD` chain and the DUT is configured as router in order to forward all traffic received from one interface to the other. The generated traffic is uniformly distributed among all the rules[7], without any packet hitting the default rule. Since each rule is a 5-tuple, the number of TCP generated flows is equal to the number of rules.

Evaluation metrics. We report the UDP throughput (in Mpps) averaged among 10 different runs. This value is taken by adjusting the sending rate not to exceed 1% packet loss. Single-core results are taken by setting the interrupts mask of each ingress receive queue to a single core, while multi-core performance (14 cores in our case) represent the standard case where all the available cores in the DUT are used.

Results. Figure 5.6a and 5.6b show respectively the single-core and multi-core forwarding performance results. We can notice from Figure 5.6a how `bpf-iptables` outperforms `iptables` by a factor of two even with a small number of rules (i.e.,

[7]We used a customized version of Pktgen-DPDK [110] to randomly generate packet for a given range of IPv4 addresses and L4 port values.

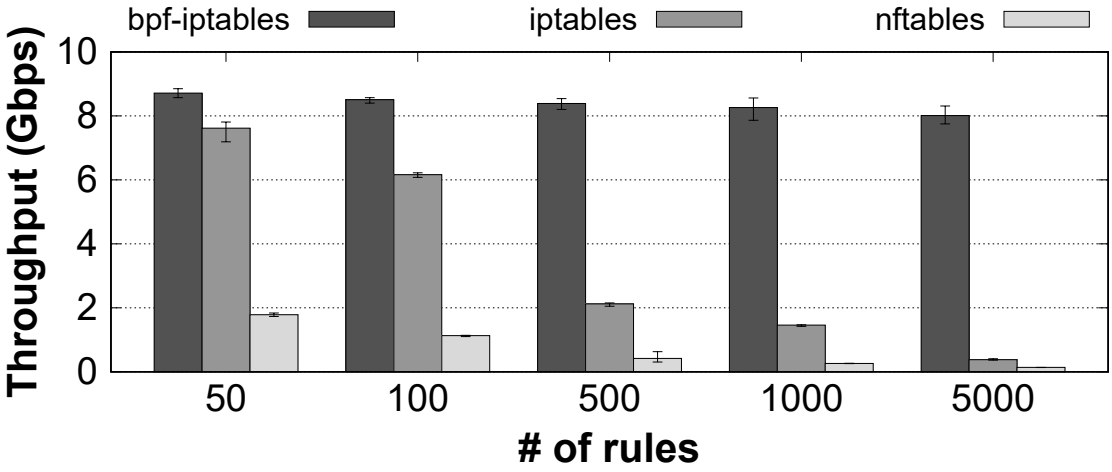


Figure 5.7: Performance of the `INPUT` chain with an increasing number of rules. `bpf-iptables` runs on a single CPU core and `iperf` on another core.

50); this gap is even larger with `nftables`, which is almost 5 times slower in the same conditions. The advantage of `bpf-iptables` is even more evident with more rules; the main performance bottleneck is the scanning of the entire bitvector in order to find the final matching rule, whose size depends on the number of rules (Section 5.4.4). Finally, Figure 5.6b shows how `bpf-iptables` scale across multiple cores; the maximum throughput is achieved with 1K rules since the number of generated flows with a smaller number of rules is not enough to guarantee uniform processing across multiple cores (due to the RSS/RFS feature of the NIC), with a resulting lower throughput.

5.6.2.3 Performance dependency on the number of rules (`INPUT` chain)

This test loads all the rules in the `INPUT` chain; traffic traverses the firewall and terminates on a local application, hence following the same path through the TCP/IP stack for all the firewalls under testing. As consequence, any performance difference is mainly due to the different classification algorithms. We used `iperf` to generate UDP traffic (using its default packet size for UDP) toward the DUT, where the default `accept` policy causes all packet to be delivered to the local `iperf` server, where we compute the final throughput. To further stress the firewall, we used eight parallel `iperf` clients to generate the traffic, saturating the 40Gbps link.

Evaluation metrics. We report the UDP throughput (in Gbps) among 10 different runs; we forced the firewall to run on a single core, while the `iperf` server runs on a different core.[8]

[8]We set the affinity of iperf on a single core and then we forced all interrupts on another one.

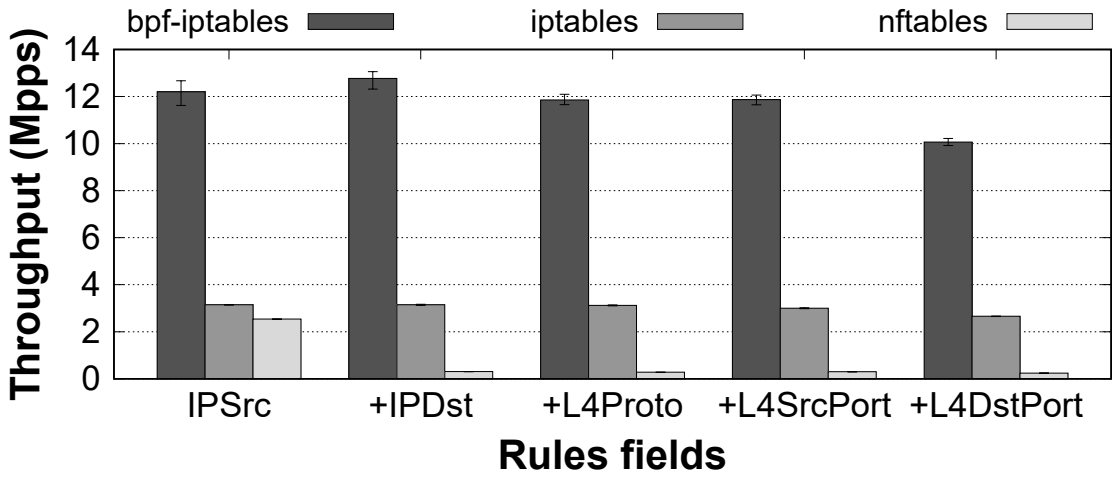


Figure 5.8: Multi-core performance comparison when varying the number of fields in the rulesets. Generated traffic (64B UDP packets) is uniformly distributed among all the rules.

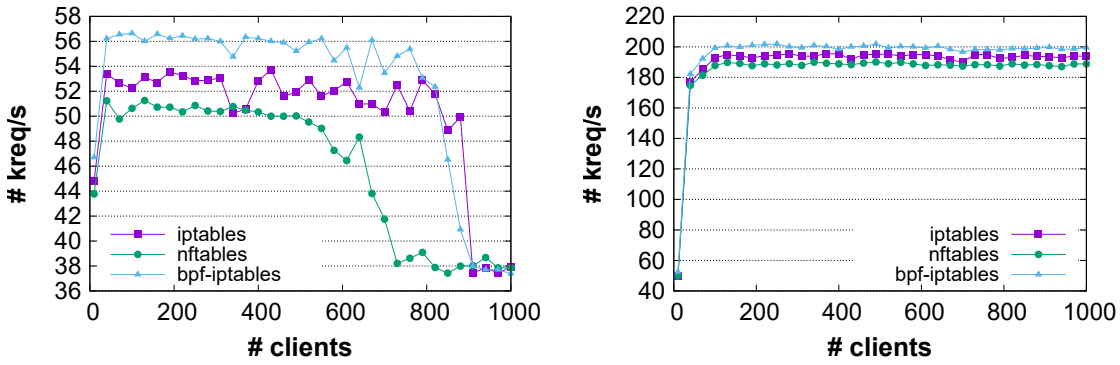
Results. Figure 5.7 shows how `bpf-iptables` perform better than the other firewalls, with an increasing gap with larger rulesets. However, the advantage with a low number of rules is smaller compared to the previous case; in fact, in this scenarios, `bpf-iptables` cannot avoid the cost of passing through the TCP/IP stack (and the allocation of the `sk_buff`). Therefore its performance advantage is given only by the different classification algorithm, which is more evident when the number of rules grows. However, it is important to note that in case of `DROP` rules, `bpf-iptables` discards the packets far before they reach the local application, with a sensible performance advantage thanks to the early processing of XDP.

5.6.2.4 Performance dependency on the number of matching fields

Since the `bpf-iptables` modular pipeline requires a separate eBPF program (hence an additional processing penalty) for each matching field, this test evaluates the throughput of `bpf-iptables` when increasing the number of matching fields in the deployed rules in order to characterize the (possible) performance degradation when operating on a growing number of protocol fields.

Ruleset. We generated five different rulesets with a fixed number of rules (i.e., 1000) and with an increasing complexity that goes from matching only the srcIP address to the entire 5-tuple. All the rules have been loaded in the `FORWARD` chain and have the `ACCEPT` action, while the default action of the chain is `DROP`.

Test setup and evaluation metrics. Same as Section 5.6.2.2.



(a) NIC interrupts set to a single core; `nginx` running on the remaining ones.

(b) NIC interrupts are set to all the cores; `nginx` running without any restrictions.

Figure 5.9: Connection tracking with an increasing number of clients (number of successfully completed requests/s).

Results. Results in Figure 5.8 show that `iptables` performs almost the same independently on the complexity of the rules; this is expected given that is cost is dominated by the number of rules. Results for `bpf-iptables` are less obvious. While, in the general case, increasing the number of fields corresponds to a decrease in performance (e.g., rules operating on the 5-tuple show the lowest throughput), this is not always true, with the first four columns showing roughly the same value and the peak observed when operating on two fields. In fact, the performance of `bpf-iptables` are influenced also by the *type* of field and *number* of values for each field. For instance, the matching against IP addresses requires, in the general case, a longest prefix match algorithm; as consequence, `bpf-iptables` uses an `LPM_TRIE`, whose performance depend on the number of distinct values. In this case, a single matching on a bigger LPM_TRIE results more expensive than two matches on two far smaller LPM_TRIE, which is the case when rules operate on both IP source and destination addresses[9].

5.6.2.5 Connection Tracking Performance

This test evaluates the performance of the connection tracking module, which enables stateful filtering. We used HTTP traffic to stress the rather complex state machine of that protocol (Section 5.4.5) by generating a high number of *new* connections per second, taking the number of successfully completed sessions as performance indicator.

[9]First ruleset had 1000 rules, all operating on source IP addresses. Second ruleset used #50 distinct srcIPs and #20 distinct dstIPs, resulting again in 1000 rules.

Test setup. In this test `weighttp` [10] generated 1M HTTP requests towards the DUT, using an increasing number of concurrent clients to stress the connection tracking module. At each request, a file of 100 byte is returned by the `nginx` web server running in the DUT. Once the request is completed, the current connection is closed and a new connection is created. This required to increase the limit of 1024 open file descriptors per process imposed by Linux in order to allow the sender to generate a larger number of new requests per second and to enable the `net.ipv4.tcp_tw_reuse` flag to reuse sessions in `TIME_WAIT` state in both sender and receiver machines[10].

Ruleset. This ruleset is made by three rules loaded in the `INPUT` chain, hence operating only on packets directed to a local application. The first rule *accepts* all packets belonging to an `ESTABLISHED` session; the second rule *accepts* all the `NEW` packets coming from the outside and with the TCP destination port equal to 80; the last rule *drops* all the other packets coming from outside.

Evaluation metrics. We measure the number of successfully completed requests; in particular, `weighttp` increments the above number only if a request is completed within 5 seconds.

Results. `bpf-iptables` scores better in both single-core and multi-core tests, with `iptables` performing from 5 to 3% less and `nftables` being down from 7 to 10%, as shown in Figures 5.9a and 5.9b. However, for the sake of precision, the connection tracking module of `bpf-iptables` does not include all the features supported by `iptables` and `nftables` (Section 5.4.5). Nevertheless, we remind that this logic can be customized at run-time to fit the necessity of the particular running application, including only the required features, without having to update the Linux kernel.

5.6.3 Realistic Scenarios

In this set of tests we analyzed some scenarios that are common in enterprise environments, such as *(i)* protecting servers in a DMZ, and *(ii)* performance under DDoS attack.

5.6.3.1 Enterprise public servers

This test mimics the configuration of an enterprise firewall used as *front-end* device, which controls the traffic directed to a protected network (e.g., `DMZ`) that

[10]We also tuned some parameters (e.g., max backlog, local port range) in order to reduce the overhead of the web server.

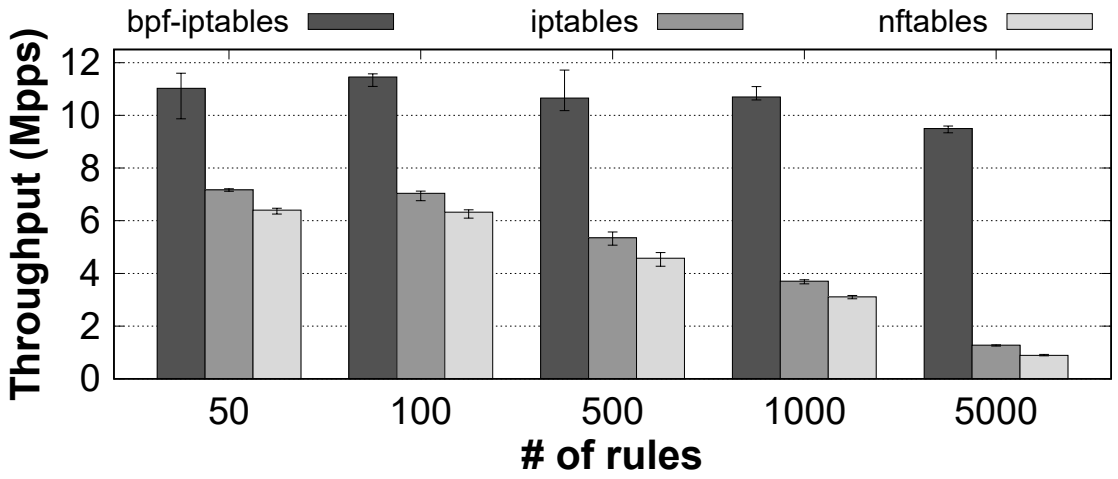


Figure 5.10: Throughput when protecting a variable number of services within a DMZ. Multi-core tests with UDP 64B packets, bidirectional flows.

hosts a set of servers that must be reachable from the outside world. We increase the number of public servers that needs to be protected, hence tests were repeated with different number of rules.

Ruleset. The first rule *accepts* all the `ESTABLISHED` connections towards the protected network; then, a set of rules *accept* `NEW` connections generated by the servers in the protected network towards the outside world; the latest set of rules enable the communication towards the services exposed in the protected network by matching on the destination IP, protocol and L4 port destination of the incoming packets. Among the different runs we used an increasing number of rules ranging from 50 to 5K, depending on the number of public services that are exposed to the outside world.

Test setup. All the rules are loaded in the `FORWARD` chain and the traffic is generated so that the 90% is evenly distributed among all the rules and the 10% matches the default `DROP` rule. The packet generator is connected to the DUT through two interfaces, simulating a scenario where the firewall is between the two (public and protected) networks. When traffic belonging to a specific flow is seen in both directions, the session is considered `ESTABLISHED` and then will match the first rule of the ruleset.

Evaluation metrics. The test has been repeated 10 times; results report the throughput in Mpps (for 64B UDP packets).

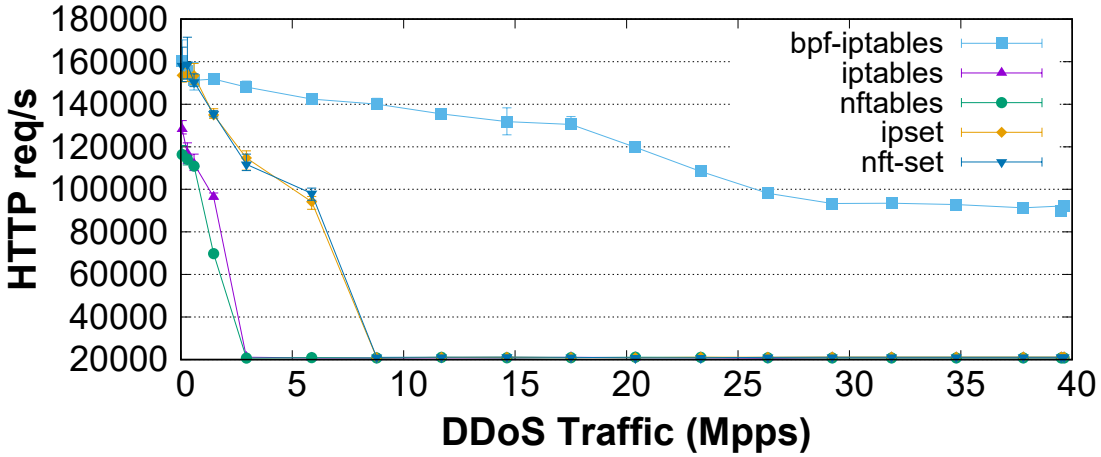


Figure 5.11: Multi-core performance under DDoS attack. Number of successful HTTP requests/s under different load rates.

Results. `bpf-iptables` outperforms existing solutions thanks to the optimized path for the `FORWARD` chain, which transparently avoids the overhead of the Linux TCP/IP stack, as shown in Figure 5.10. In addition, its throughput is almost independent from the number of rules thanks to the optimization on the `ESTABLISHED` connections (Section 5.4.4.2), which avoids the overhead of the classification pipeline if the conntrack module recognizes an `ESTABLISHED` connection that should be *accepted*. Even if `iptables` would also benefit from the fact that most packets match the first rule, hence making the linear search faster, the overall performance in Figure 5.10 show a decrease in throughput when the number of rules in the ruleset grows. This is primarily due to the overhead to recognize the traffic matching the default rule (`DROP` in our scenario), which still requires to scan (linearly) the entire ruleset.

5.6.3.2 Performance under DDoS Attack

This tests evaluates the performance of the system under DDoS attack. We analyzed also two optimized configurations of `iptables` and `nftables` that make use of `ipset` and `sets` commands, which ensures better performance when matching an entry against a set of values.

Ruleset. We used a fixed set of rules (i.e., 1000) matching on IP source, protocol and L4 source port, `DROP` action. Two additional rules involve the connection tracking to guarantee the reachability of internal servers; (*i*) *accepts* all the `ESTABLISHED` connections and (*ii*) *accepts* all the `NEW` connection with destination L4 port 80.

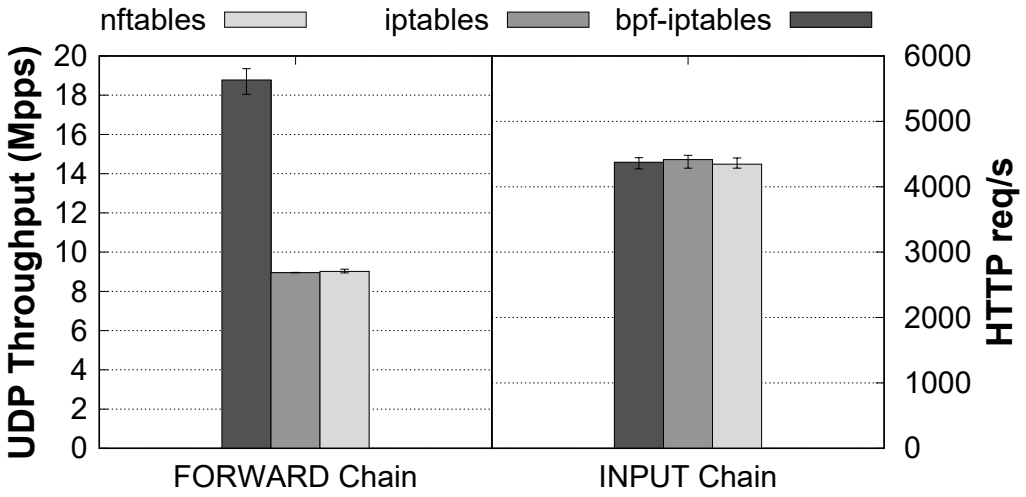


Figure 5.12: Performance with single default `ACCEPT` rule (baseline). Left: UDP traffic, 64B packets matching the `FORWARD` chain. Right: number of HTTP requests/s (downloading a $1MB$ web page), TCP packets matching the `INPUT` chain.

Test setup and evaluation metrics. The packet generator sends 64Bytes UDP packets towards the server with the same set of source IP addresses and L4 ports configured in the blacklisted rules. DDoS traffic is sent on a first port connected to the DUT, while a `weighttp` client sends traffic on a second port, simulating a legitimate traffic towards a `nginx` server running in the DUT. `Weighttp` generates 1M HTTP requests using 1000 concurrent clients; we report the number of successfully completed requests/s, with a timeout of 5 seconds, varying the rate of DDoS traffic.

Results. Figure 5.11 shows that the performance of `bpf-iptables`, `ipset` and `nft-set` are similar for of low-volume DDoS attacks; `iptables` and `nftables` are slightly worse because of their inferior matching algorithm. However, with higher DDoS load (> 8Mpps), the performance of `ipset` and `nft-set` drop rapidly and the server becomes unresponsive, with almost no requests served; `iptables` and `nftables` are even worse (zero goodput at 2.5Mpps). Vice versa, thanks to its matching pipeline at the XDP level, `bpf-iptables` can successfully sustain ~95.000 HTTP requests/s of legitimate traffic when the DDoS attack rate is more than 40Mpps, i.e., ~60% of the maximum achievable load. Higher DDoS load was not tested because of a limitation of our traffic generator.

5.6.4 Microbenchmarks

5.6.4.1 Baseline performance

This test analyzes the overhead of `bpf-iptables` on a vanilla system, without any firewall rule. This represents the most favorable case for `iptables` where cost grows linearly with the number of rules, while `bpf-iptables` has to pay the cost of some programs at the beginning of the pipeline that must be always active, such as the connection tracking and the logic that applies the default action to all packets (i.e., `ALLOW`). The left side of Figure 5.12 shows the performance of `bpf-iptables`, `iptables` and `nftables` when the traffic (64B UDP packets) traverses the `FORWARD` chain. This case shows a considerable advantage of `bpf-iptables` thanks to its optimized forwarding mechanism (Section 5.4.4.2). The situation is slightly different when the traffic hits the `INPUT` chain (Figure 5.12, right). In fact, in such case the packets has to follow the usual path towards the stack before reaching the local application, with no chance to shorten its journey. While `bpf-iptables` does not show the advantages seen in the previous case, it does not show any worsening either, hence demonstrating that the overhead of the running components is definitely limited.

5.6.4.2 Rules insertion time

The LBVS matching algorithm requires the update of the entire pipeline each time the ruleset changes (Section 5.4.4.3). This test evaluates the time required to insert the $(n+1)^{\text{th}}$ rule when the ruleset already contains n rules; in case of `iptables` and `nft`, this has been measured by computing the time required to execute the corresponding userspace tool. Results, presented in Table 5.1, show that both `iptables` and `nftables` are very fast in this operation, which completes in some tens of milliseconds; `bpf-iptables`, instead, requires a far larger time (varying from 1 to 2.5s with larger rulesets). To understand the reason of this higher cost, we exploded the `bpf-iptables` rules insertion time in three different parts. Hence, `t1` indicates the time required by the `bpf-iptables` control plane to compute all the value-bitvector pairs for the current ruleset. Instead, `t2` indicates the time required to compile and inject the new eBPF classification pipeline in the kernel; during this time, `bpf-iptables` continues to process the traffic according to the old ruleset, with the *swapping* performed only when the new pipeline is ready[11]. Finally, `t3` is the time required to delete the old chain, which has no impact on the user experience as the new pipeline is already filtering traffic after `t2`. Finally,

[11]Since time `t2` depends on the number of matching fields required by each rule (`bpf-iptables` instantiates the minimum set of eBPF programs necessary to handle the current configuration), numbers in Table 5.1 take into account the worst case where all the rules require matching on all the supported fields.

Table 5.1: Comparison of the time required to append the $(n+1)^{\text{th}}$ in the ruleset in milliseconds (ms).

# rules	iptables	nftables	bpf-iptables			HORUS	
			t1[1]	t2[2]	t3[3]	t_H1[4]	t_H2[5]
0	15	31	0.15	1165	0.34	382	0.0024
50	15	34	2.53	1560	0.36	1.08	0.0026
100	15	35	5.8	1925	0.35	2.06	0.0026
500	16	36	17	1902	0.34	8.60	0.0027
1000	17	69	33.4	1942	0.34	14.4	0.0027
5000	28	75	135	2462	0.38	37.3	0.0031

[1] Time required to compute all the bitvectors-pairs.
[2] Time required to create and load the new chain.
[3] Time required to remove the old chain.
[4] Time required to identify the rules belonging to a HORUS set.
[5] Time required to insert the new rule in the HORUS set.

the last column of Table 5.1 depicts the time required to insert a rule handled by HORUS (Section 5.4.4.2). Excluding the first entry of this set that requires to load the HORUS eBPF program, all the other entries are inserted in the HORUS set within an almost negligible amount of time (t_H2). Instead, the detection if the new rule belongs to an HORUS set takes more time (t_H1 ranges from 1 to 40ms), but this can be definitely reduced with a more optimized algorithm.

5.6.4.3 Ingress pipeline: XDP vs. TC

bpf-iptables attaches its ingress pipeline on the XDP hook, which enables traffic processing as early as possible in the Linux networking stack. This is particularly convenient when the packet matches the DROP action or when we can bypass the TCP/IP stack and forward immediately the packet to the final destination (*optimized forwarding*, Section 5.4.4.2). However, when an eBPF program is attached to the XDP hook, the Generic Receive Offload[12] feature on that interface is disabled; as a consequence, we may incur in higher processing costs in presence of large TCP *incoming* flows. Results in Figure 5.13, which refer to a set of parallel TCP flows

[12]Generic Receive Offload (GRO) is a software-based offloading technique that reduces the per-packet processing overhead by reassembling small packets into larger ones.

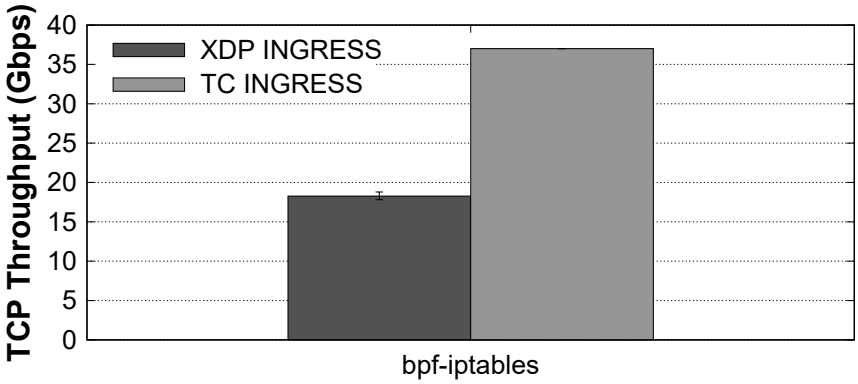


Figure 5.13: TCP throughput when the `bpf-iptables` ingress pipeline (with zero rules) is executed on either `XDP` or `TC` ingress hook; `bpf-iptables` running on a single CPU core; `iperf` running on all the other cores.

between the traffic generator and the DUT, with a void `INPUT` chain and the default `ACCEPT` action, show clearly how the `XDP` ingress pipeline pays a higher cost compared to `TC`, which easily saturates our 40Gbps link[13]. This higher cost is given by the larger number of (small) packets to be processed by `bpf-iptables` because of the lack of GRO aggregation; it is important to note that this cost is not present if TCP data exits from the server (*outgoing* traffic), which is a far more common scenario.

5.7 Additional Discussion

Although one of the main assumption of our work was to rely on a vanilla Linux kernel (Section 5.2.4), as a possible future work we present here a set of viable kernel modifications that are compatible with `bpf-iptables` and that may enable new optimizations.

New kernel hooks. Being based on eBPF, `bpf-iptables` uses a different set of hooks compared to the ones used by the `netfilter` subsystem (Section 5.2.1). This introduces the need to predict, in a preceding eBPF hook, some decisions that would be performed only later in the Linux stack. A possible alternative consists in adding new eBPF hooks that operate in `netfilter`, hence enabling the replacement of selected portions of the above framework that suffer more in terms of performance (e.g., `iptables` classification pipeline), while reusing existing

[13]To avoid TCP and application-level processing to become the bottleneck, we set all the NIC interrupts to a single CPU core, on which `bpf-iptables` has to be executed, while `iperf` uses all the remaining ones.

and well-tested code (e.g., `netfilter` conntrack). Although this would be the most suitable choice for a 100% `iptables`-compatible eBPF-based firewall, on the other side it would unavoidably limit the overall performance of the system. In fact, this would set the baseline performance of `bpf-iptables` to the one of the corresponding TCP/IP stack layer, because of the large amount of code shared between the two approaches and the impossibility to leverage earlier processing provided by the XDP hook. Moreover, the early packet steering provided by XDP enables also the creation of the exact processing pipeline that is required in any given moment in time, instantiating only the proper eBPF modules. This would avoid any source of overhead in the processing path of a packet, which would not be possible in case existing kernel network stack components are used.

New eBPF helpers. Adding new eBPF helpers is definitely a suitable direction, in particular with respect to our eBPF conntrack (Section 5.4.5) that is far from complete and supports only basic scenarios. A dedicated helper would enable a more complete implementation without having to deal with the well-known limitations of eBPF programs (e.g., number of instructions, loops). A similar helper that reused the `netfilter` connection tracking was proposed in [158], which was at the foundation of an alternative version of `bpf-iptables` [111]. However, based on the above prototype, we would suggest a custom implementation of the conntrack module in order to be independent from the network stack; the above implementation assumed the use of `sk_buff` structure and hence was available only to eBPF programs attached to the TC hook.

Improve eBPF internals. One of the biggest limitation of the eBPF subsystem that we faced in this work is the maximum number of allowed instructions, currently constrained to 4K, which limited the maximum number of supported rules to ~8K (Section 5.4.4, without `HORUS`). In this respect, the extension of the eBPF verifier (available for kernel v5.3+) to support *bounded loops* would be extremely helpful [136]. At the same way, a recent patch [150] that introduced the support for larger eBPF programs up to *one million*[14] increases the number of supported rules without any change in the overall design of the system.

5.8 Conclusions

This Chapter presents `bpf-iptables`, an eBPF-based Linux firewall designed to preserve the `iptables` filtering semantic while improving its speed and scalability, in particular when a high number of rules are used. Being based on eBPF,

[14]This value indicates the number of instructions processed by the verifier.

`bpf-iptables` is able to take advantage of the characteristics of this technology, such as the dynamic compilation and injection of the eBPF programs in the kernel at run-time in order to build an optimized data-path based on the actual firewall configuration. The tight integration of `bpf-iptables` with the Linux kernel may represent a great advantage over other solutions (e.g., DPDK) because of the possibility to cooperate with the rest of the kernel functions (e.g., routing) and the other tools of the Linux ecosystem. Furthermore, `bpf-iptables` does not require custom kernel modules or additional software frameworks that could not be allowed in some scenarios such as public data-centers.

`Bpf-iptables` guarantees a huge performance advantage compared to existing solutions, particularly in case of an high number (i.e., up to 32K in the current prototype) of filtering rules; furthermore, it does not introduce undue overheads in the system when no rules are instantiated, even though in some cases the use of `XDP` on the ingress hook could hurt the overall performance of the system. Existing eBPF limitations have been circumvented with ad-hoc engineering choices (e.g., classification pipeline) and clever optimizations (e.g., HORUS), which guarantee further scalability and fast update time.

On the other hand, currently `bpf-iptables` supports only a subset of the features available in `netfilter`-based firewalls. For instance, `iptables` is often used to also handle *natting* functions, which we have not considered in this Chapter, as well as the features available in `ebtables` and `arptables`. Those functionality, together with the support for additional matching fields are considered as possible direction for our future work.

Chapter 6

Introducing SmartNICs in Server-based Data Plane Processing: the DDoS Mitigation Use Case

6.1 Introduction

With the objective of further reducing the workload on the precious general-purpose CPU cores of the servers, system administrators have recently resumed the old idea of introducing *programmable* intelligent networking adapters (a.k.a., *SmartNICs*) in their servers [156, 103, 64, 41], combining the flexibility of software network functions with the improved performance of the hardware NIC acceleration. Smart Network Interface Cards (SmartNIC) offer hardware accelerators that enable to partially (or fully) offload packet processing functions; examples include load balancing [116], key-value stores [145] or more generic flow-level network functions [128, 118]. On the other hand, SmartNICs may present additional challenges due to their limited memory and computation capabilities compared to current high-performance servers.

In this Chapter we explored the potential of exploiting SmartNICs on a specific use case, i.e., to mitigate volumetric DDoS attacks, which are considered as one of the major threats in today's Internet, accounting for the 75.7% of the total DDoS attacks [5, 149, 4]. While the *detection* of DDoS attacks is a largely studied problem in the literature with several algorithms proposed to rapidly and efficiently detect an ongoing attack, in this Chapter we focus on the challenges related to the DDoS attack *mitigation*; in particular, we explore how the recent advances on the host data-plane acceleration can be used to adequately handle the large speeds required by today's networks. This work results particularly beneficial also for the

application showed in Chapter **??**, where the `bpf-iptables` classification pipeline can be enhanced and improved with the hardware filtering mechanism available in the SmartNIC.

We provide the following contributions. First, we analyze the various approaches that can be used to design an efficient and cost-effective DDoS mitigation solution. As generally expected, our results show that offloading the mitigation task to the programmable NIC yields significant performance improvements; however, we demonstrate also that, due to the memory and compute limitations of current SmartNIC technologies, a fully offloaded solution may lead to deleterious performance. Second, as a consequence of the previous findings, we propose the design and implementation of a hybrid mitigation pipeline architecture that leverages the flexibility of eBPF/eXpress Data Path (XDP) to handle different type of traffic and attackers and the efficiency of the hardware-based filtering in the SmartNIC to discard traffic from malicious sources. Third, we present a mechanism to transparently offload part of the DDoS mitigation rules into the SmartNIC, which takes into account the most aggressive sources, i.e., the ones that largely impact on the mitigation effectiveness.

The rest of the Chapter is structured as follows. Section 6.2 presents a high-level overview of the SmartNIC and TC Flower, the flow classifier of the Linux traffic control kernel subsystem. Section 6.3 analyzes the different approaches that can be used to build an efficient DDoS mitigation solution. Section 6.4 presents the design of an architecture that uses the above mentioned technologies to both detect and mitigate DDoS attacks, including the offloading algorithm adopted to install the rules into the SmartNIC (Section 6.4.1.1), while keeping the flexibility and improved performance of the in-kernel XDP packet processing. Finally, Section 6.5 provides the necessary evidence to the previous findings, Section 6.6 briefly discusses the related works and Section 6.7 concludes the Chapter.

6.2 Background

6.2.1 SmartNICs

Smart Network Interface Cards (SmartNICs) are intelligent adapters used to boost the performance of servers by offloading (part of) the network processing workload from the host CPU to the NIC itself [154]. Although the term SmartNIC is being widely used in the industry and academic world, there is still some confusion over the precise definition. We consider traditional NICs the devices that provide several pre-defined offloaded functions (e.g., transmit/receive segmentation offload, checksum offload) without including a fully programmable processing path, e.g., which may involve the presence of a general-purpose CPU on board. In our context, a *SmartNIC* is a NIC equipped with a fully-programmable system-on-chip

(SoC) multi-core processor that is capable to run a fully-fledged operating system, offering more flexibility and hence potentially taking care of any arbitrary network processing task. This type of SmartNIC can also be enhanced with a set of specialized hardware functionalities that can be used to accelerate specific class of functions (e.g., OpenvSwitch data-plane) or to perform generic packet and flow-filtering. On the other hand, they have limited compute and memory capabilities, making not always possible (or efficient) to completely offload all types of tasks. Furthermore, SmartNICs feature their own operating system and therefore may have to be handled separately from the host. For instance, offloading a network task to the SmartNIC may require the host to have multiple interactions with the card, such as to compile and inject the new eBPF code, to execute additional commands (either on the host, or directly on the card) to exploit the available features such as configure hardware co-processors. Finally, no current standard exist to interact with SmartNICs, hence different (and often proprierary) methods have to be implemented when the support of several manufacturers is required.

6.2.2 TC Flower

The Flow Classifier is a feature of the Linux Traffic Control (TC) kernel subsystem that provides the possibility to match, modify and apply different actions to a packet based on the flow it belongs to. It offers a common interface for hardware vendors to implement an offloading logic within their devices; when a TC Flower rule is added, active NIC drivers check if that rule is supported in hardware; in that case the rule is pushed to the physical card, causing packets to be directly matched in the hardware device [77], hence resulting in greater throughput and a decrease of the host CPU usage. TC Flower represents a promising technology that can hide the differences between different hardware manufacturers, but it not able (yet) to support all the high-level features that may be available in modern SmartNICs.

6.3 DDoS Mitigation: Approaches

Once a DDoS attack is detected, efficient packet dropping is a fundamental part of a DDoS attack mitigation solution. In a typical DDoS mitigation pipeline, a set of mitigation rules are deployed in the server's data plane to filter the malicious traffic. The strategy used to block the malicious sources may be determined by several factors such as the characteristics of the server (e.g., availability of a SmartNIC, its hardware capabilities), the characteristics of the malicious traffic (e.g., number of attackers) or the type and complexity of the rules that are used to classify the illegitimate traffic. In particular, we envision the following three approaches.

6.3.0.1 Host-based mitigation

In this case all traffic (either malicious or legitimate) is processed by the host CPU, which drops incoming packets that match a given blacklist of malicious sources; this represents the only viable option if the system lacks of any underlying hardware accelerators/offloading functionalities. All the host-based mitigation techniques and tools used today fall in two different macro-categories depending on whether packets are processed at kernel or user-space level.

Focusing on Linux-based system, the first category includes `iptables` and its derivatives, such as `nftables`, which represent the main tools used to mitigate DDoS attacks. It allows to express complex policies to the traffic, filtering packets inside the `netfilter` subsystem. However, the deep level in the networking stack where the packet processing occurs causes poor performance when coping with increasing speed of the today's DDoS attacks, making this solution practically unfeasible, as demonstrated in Section 6.5. As opposite to kernel-level processing, a multitude of fast packet I/O frameworks relying on specialized NIC/networking drivers and user-space processing have been built over the past years. Examples such as Netmap [133], DPDK [54], PF_RING ZC [121] rely on a small kernel component that maps the NIC device memory directly to user space, hence making it directly available to (network-specialized) userland applications instead of relying on normal kernel data-path processing. This approach provides huge performance benefits compared to the standard kernel packet processing but incurs in several non-negligible drawbacks. First of all, these frameworks require to take the exclusive ownership of the NIC, so that all packets received are processed by the userspace application. This means that, in a DDoS mitigation scenario, packets belonging to legitimate sources have to be inserted back into the kernel, causing unnecessary packet copies that slow down the performance[1]. Furthermore, these frameworks require the fixed allocation of one (or more) CPU cores to the above programs, independently from the presence of an ongoing attack, hence reducing the performance-cost ratio, as precious CPU resources are no longer available for normal processing tasks (e.g., virtual machines).

eBPF/XDP can be considered as a mix of the previous approaches. It is technically a kernel-space framework, although XDP programs can be injected from userspace to the kernel, after guaranteeing that all security properties are satisfied. XDP programs are executed in the kernel context but as early as possible, well before the `netfilter` framework, hence providing an improvement of an order of magnitude compared to `iptables`. The adoption of XDP to implement packet filtering functionalities has grown over the years; *(i)* its perfect integration with the

[1]It is worth mentioning that Netmap has a better kernel integration compared to DPDK; in fact, it is possible to inject packets back into the kernel by just passing a pointer, without any copy. However, it is still subjected to a high CPU consumption compared to eBPF/XDP.

Linux kernel makes it more efficient to pass legitimate packets up to the stack, *(ii)* its simple programming model makes it easy to express customized filtering rules without taking care of low-level details such as required by common user-space framework and *(iii)* its event-driven execution gives the possibility to consume resources only when necessary, providing a perfect trade-off between performance and CPU consumption.

6.3.0.2 SmartNIC-based mitigation

If the server is equipped with a SmartNIC, an alternative approach would be to offload the entire mitigation task to this device. This enables to dedicate all the available resources on the host CPU to the target workloads, operating only on the legitimate traffic, freeing the host CPU from spending precious CPU cycles in the mitigation. However, although SmartNICs (by definition) support arbitrary data path processing, they often differ on how this can be achieved. Possible options range from running a custom executable, which should already be present on the card, to dynamically inject a new program created on the fly, e.g., thanks to technologies such as XDP or P4, or to directly compile those programs into the hardware device [34]. This makes more cumbersome the implementation of offloading features that run on cards from multiple manufacturers.

In our context, we envision two different options: (*i*) exploit any hardware filter (if available) in the SmartNIC and, if the number of blacklisted addresses exceeds the capability of the hardware (which may be likely, given the typical size of the above structure), block the rest of the traffic with a custom dropping program (e.g., XDP) running on the NIC CPU; (*ii*) block all the packets in software, running entirely on the SmartNIC CPU, e.g., in case the card does not have any hardware filtering capability. In both cases, the surviving (benign) traffic is redirected to the host where the rest of server applications are running. An evaluation of the above possibilities will be carried out in Section 6.5.

6.3.0.3 Hybrid (SmartNIC + XDP Host)

An alternative strategy that combines the advantages of the previous approaches would be to adopt a hybrid solution where part of the malicious traffic is dropped by the SmartNIC (reducing the overhead on the host's CPU) and the remaining part is handled on the host, possibly leveraging the much greater processing power available in modern server CPUs compared to the one available in embedded devices.

In this scenario, we exploit the fixed hardware functions commonly available in the current SmartNICs to perform stateless matching on selected packet fields and apply simple actions such as modify, drop or allow packets. To avoid redirecting all the traffic to the (less powerful) SmartNIC CPU, we could let it pass through the

above hardware tables (where the match/drop is performed at line rate) and forward the rest of the packets to the host, where the remaining part of the mitigation pipeline is running. However, given the limited number of entries often available in the above hardware tables, which are not enough to contain the large number of mitigation rules needed during a large DDoS attack, the whole list of dropping targets is partitioned between the NIC and the host dropping program (e.g., XDP). This requires specific algorithms to perform this splitting, which should keep into account the difference in terms of supported rules and their importance. Interesting, this scenario in which the companion filtering XDP program is executed in the server is also compatible with some traditional NICs that support fixed hardware traffic filtering, such as Intel cards with Flow Director[2]. In this case, the mitigation module can use the card-specific syntax (e.g., Flow Director commands) to configure filtering rules, with the consequent decrease of the filtering processing load in the host.

6.4 Architecture and Implementation

This section presents a possible architecture that can be used to compare the previous three approaches in the important use case of the DDoS mitigation, enabling a fair comparison of their respective strength and weaknesses in the implementation of an efficient and cost-effective mitigation pipeline. In particular, we present the different components constituting the proposed architecture (shown in Figure 6.1) and their role, together we some implementation details that result from the use of the assessed technologies.

6.4.1 Mitigation

The first program encountered in the pipeline is the *filtering* module, which matches the incoming traffic against the list of blacklisted entries to drop packets coming from malicious sources; surviving packets are redirected to the host where additional (more advanced) checks can be performed before redirecting packets directly to the next program in the pipeline (i.e., the *feature extraction*).

Although our architecture is flexible enough to instantiate the filtering program in different locations (e.g., SmartNIC, Host, and even partitioned across the two above), at the beginning we instantiate an XDP *filtering* program in the host in order to obtain the necessary traffic information and decide the best mitigation strategy. If the userspace DDoS *mitigation* module recognizes the availability of the hardware offload functionality in the SmartNIC, it starts adding the filtering

[2]The Flow Director is an Intel feature that supports advanced filters and packet processing in the NIC; for this reason it is often used in scenarios where packets are small and traffic is heavy (e.g., DoS attacks).

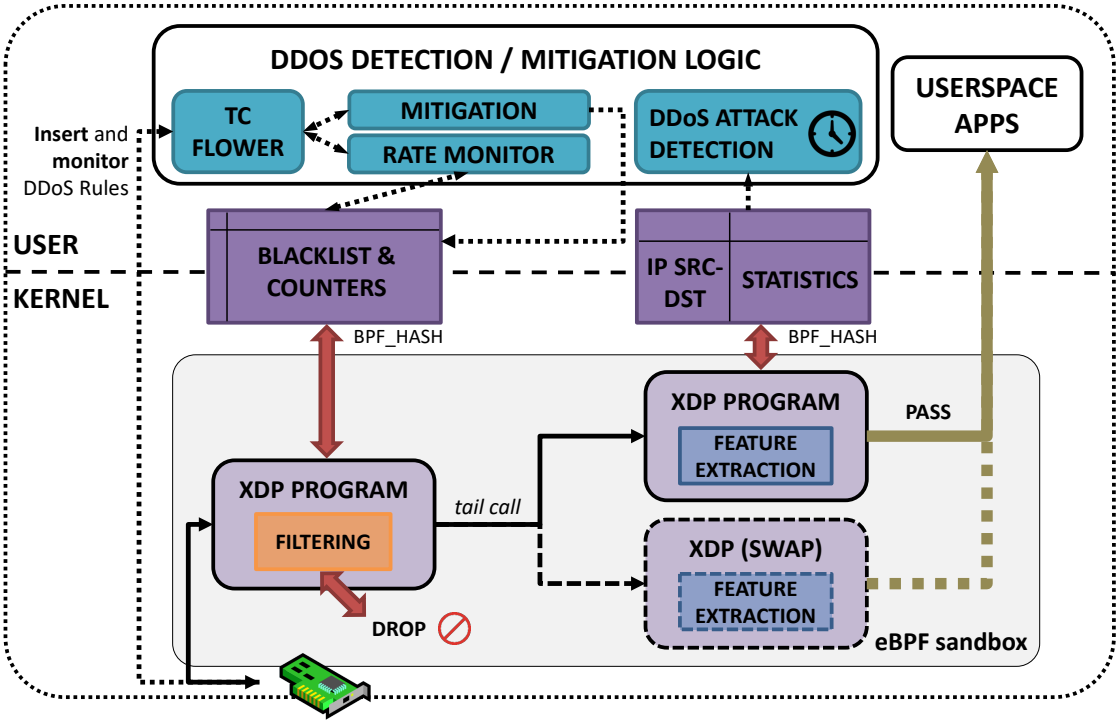


Figure 6.1: High-level architecture of the system.

rules into the hardware tables, causing malicious packet to be immediately dropped in hardware. However, since those tables have often a limited size (typically ∼1-2K entries), we place the most active top-K malicious talkers in the SmartNIC hardware tables, where K is the size of those tables, while the remaining ones are filtered by the XDP program running either on the SmartNIC CPU or on the host, depending on a configuration option that enables us to compare the results with different operating conditions.

6.4.1.1 Offloading algorithm

The selection of the top-K malicious talkers that are most appropriate for hardware offloading is carried out by the *rate monitor* module, which computes a set of statistics on the dropped traffic and applies a hysteresis-based function to predict the advantages of possibly modifying the list of offloaded rules that are active in the SmartNIC. In fact, altering this list requires either computational resources or time (in our card a single rule update may require up to 2 ms), which may be unnecessary if the rank of the new top-K rules does not effectively impact on the mitigation effectiveness.

The pseudo-code of our algorithm is shown in Listing 2. First, it computes a list of the *global* top-K sources, which contains both SmartNIC and XDP entries

Algorithm 2: Offloading algorithm

Input: K, the max # of supported SmartNIC entries
Output: $\upsilon'_k \leftarrow$ The list of SmartNIC entries.

```
1  γ_k ← TOP-K Global entries
2  υ_k ← TOP-K SmartNIC entries
3  sortDescending(γ_k)
4  sortAscending(υ_k)
5  γ'_k ← γ_k - υ_k                    /* Remove already offloaded entries */
6  υ'_k ← υ_k - γ_k                    /* List of non TOP-K rules */
7  foreach γ'_{i,k} ∈ γ'_k do
8      β_i ← offloadGain(γ'_{i,k}, υ'_{i,k})
9      if β_i ≥ threshold then
10         υ'_k ← υ'_k − υ'_{i,k}      /* Remove old entry from offload list */
11         υ'_k ← υ'_k + γ'_{i,k}      /* Add new entry into offload list */
```

sorted in *descending* order according to their rate, and a second list containing only the offloaded entries, i.e., the ones present in the SmartNIC hardware tables, which is arranged in *ascending* order. Next, it computes the difference of the above lists, resulting in two lists containing two disjoint set of elements; the first list contains all the *candidate* rules that are not yet in the SmartNIC and the second list includes the SmartNIC entries that are not in the top-K anymore. At this point, starting from the first element of the former list, it calculates the possible benefit obtained by removing the first entry of the second list (given by the ratio between the rate of the two entries) and inserting this new entry in the SmartNIC; if the value is greater than a certain threshold, the entry is moved into the *offloaded* list and the algorithm continues with the next entry. This *threshold* is adjusted according to the current volume of DDoS traffic and it is inversely proportional to it; this avoids unnecessary changes in the top-K SmartNIC list when the traffic rate is low (compared to the maximum achievable rate), which may bring a negligible improvement. On the other hand, it increases the update likelihood when the volume of traffic is close to the maximum achievable rate; in this scenario, where the system is overloaded, mitigating even slightly more aggressive talkers may introduce substantial performance benefits.

6.4.2 Feature extraction

Although not strictly belonging to the mitigation pipeline, the *feature extraction* module monitors the incoming traffic and collects relevant parameters required by the mitigation algorithm (e.g., counting the number of packets for each combination of source and destination hosts). Being placed right after the mitigation module, it

receives all the (presumed) benign traffic that has not been previously dropped so that can be further analyzed and then passed up to the target applications. XDP represents the perfect technology to implement this component since it provides *(i)* the low overhead given by the kernel-level processing and *(ii)* the possibility to dynamically change the behavior of the system by re-compiling and re-injecting (in the kernel) an updated program when we require the extraction of a different set of features. Moreover, XDP offers the possibility to export the extracted information into specific key-value data structures shared between the kernel and userspace (i.e., where the DDoS attack detection algorithm is running) or to directly send the entire packet up to userspace if a more in-depth analysis is needed. In the former case, data are stored in a per-CPU eBPF hash map, which is periodically read by the userspace *attack detection* application. Since multiple instances of the same XDP program are executed in parallel on different CPU cores, each one processing a different packet, the use of a per-CPU map guarantees very fast access to data thanks to its per-core dedicated memory. As result, each instance of the *feature extraction* works independently, saving the statistics of each IP source/destination on its own private map. In the latter case, a specific eBPF helper is used to copy packets to a `perf` event ring buffer, which is then read by the userspace application.

Analysis and Aggregation. Computed traffic statistics are retrieved from each kernel-level hash-map, aggregated by the companion userspace application and saved in memory for further processing. This process was found to be relatively slow; our tests report an average of $30\mu s$ to read a single entry from the eBPF map, requiring more than ten seconds to process the entire dataset in case of large DDoS attacks (e.g., ∼300K entries). In fact, eBPF does not provide any possibility to read an entire map within a single `bpf()` system call, hence requiring to read each single value separately. As consequence, to guarantee coherent data to the userspace detection application, we should lock the entire table while reading the values, but this would result in the impossibility for the kernel to process the current incoming traffic for a considerable amount of time.

To overcome this issue, we adopted a *swappable dual-map* approach, in which the userspace application reads data from a first eBPF map that represents a snapshot of the traffic statistics at a given time, while the XDP program computes the traffic information for the incoming packets received in the the previous timespan, and saved in a second map. This process is repeated every time the periodic user-space *detection* process is triggered, allowing the detection algorithm to always work with consistent data. From the implementation point of view, we opted for a swappable *dual-program* approach instead of a swappable *dual-map* because of its reduced swapping latency. We create two *feature extraction* XDP programs, each one with its own hash-map, and swap them atomically by asking the *filtering* module to dynamically update the address of the next program in the pipeline, which basically means updating the target address of an assembly `jump` instruction.

6.4.3 Detection

The identification of a DDoS attack is performed by the *detection* module, which operates on the traffic statistics presented in the previous section and exploits the retrieved information to identify the right set of malicious sources, which are then inserted in the blacklist map used by the *filtering* module to drop the traffic.

Since the selection of the best mitigation algorithm is out of the focus here, we provide here only a small description of the possible choices that, however, need to be carefully selected depending on the characteristics of the environment and the type of workloads running on the end-hosts. Different approaches are available [149, 92] falling in two main categories: *(i)* anomaly-based detection mechanisms based on entropy [19, 18, 23], used to detect variations in the distribution of traffic features observed in consecutive timeframes and *(ii)* signature-based approaches that employ a-priori knowledge of attack signatures to match incoming traffic and detect intrusions. Of course, the type of detection algorithm may influence the exported traffic information on the *feature extraction* module; however, thanks to the excellent programmability of XDP we can change the behavior of the program without impacting on the rest of the architecture.

6.4.4 Rate Monitor

Sometimes, a given detection algorithm may erroneously detect some legitimate sources as attackers. To counter this situation, a specific mechanism is used to eliminate from the blacklist a source that is no longer considered malicious, e.g., because it was considered an attacker by mistake or because it does no longer participate to the attack. This task is performed by the *rate monitor*, which starts from the *global* list of blacklisted addresses, sorted according to their traffic volume, and examines the entries that are at the bottom of the list (i.e., the ones sending less traffic), comparing them with a threshold value; if the current transmission rate of the source under consideration is below the threshold, defined as the highest rate of packets with the same source observed under normal network activity, it is removed from the blacklist. In case the host is removed by mistake, the detection algorithm will re-add to the list of malicious sources in the next iteration.

6.5 Performance evaluation

This section provides an insight of the benefits of SmartNICs in the important use case of DDoS mitigation. First, it outlines the test environment and the evaluation metrics; then, exploiting the previously described architecture, it analyzes different approaches that exploit SmartNICs and/or other recent Linux technologies such as eBPF/XDP for DDoS mitigation, comparing with the performance achievable with commonly used Linux tools (i.e., *iptables*).

6.5.1 Test environment

Our testbed includes a first machine used as packet generator, which creates a massive DDoS attack with an increasing number of attack sources, and a second server running the DDoS mitigation pipeline. Both servers are equipped with an Intel Xeon E3-1245 v5 with a quad-core CPU @3.50GHz, 8MB of L3 cache and two 16GB DDR4-2400 RAM modules, running Ubuntu 18.04.2 LTS and kernel 4.15. The two machines are linked with two 25Gbps Broadcom Stingray PS225 SmartNICs [80], with each port directly connected to the corresponding one of the other server. We used *Pktgen-DPDK* v3.6.4 and *DPDK* v19.02 to generate the UDP traffic (with small 64B packets[3]) simulating the attack. We report the dropping rate of the system and the CPU usage, which are the two fundamental parameters to keep into account during an attack. We also measure the capability of the server to perform real work (i.e., serve web pages) while under attack, comparing the results of the different mitigation approaches. In this case, the legitimate traffic is generated using the open-source benchmarking tool `weighttp`, which creates a high number of parallel TCP connections towards the device under test; in this case we count only the successfully completed TCP sessions.

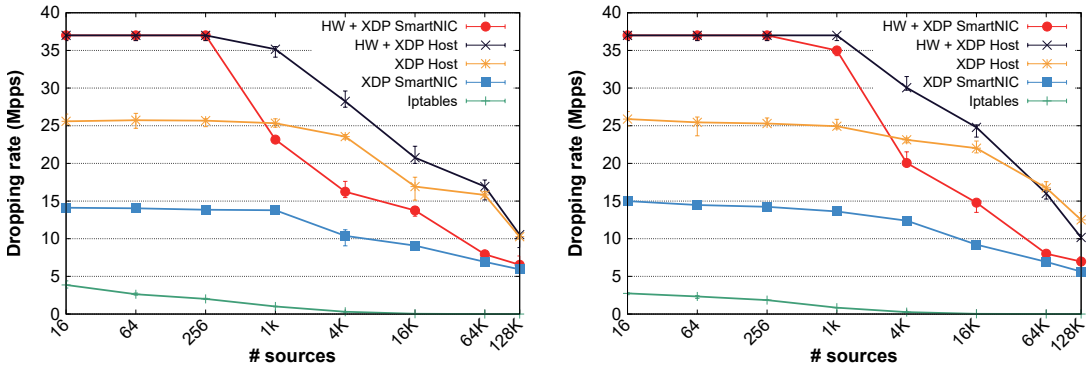
6.5.2 Mitigation performance

The first test measures the ability of the server to react to massive DDoS attacks that involve an increasing number of sources (i.e., *bots*), showing the performance of different mitigation approaches in terms of dropping rate (Mpps) and CPU consumption. We generate 64B UDP packets at line-rate at 25Gbps (i.e., 37.2Mpps); we consider both a scenario where the traffic is uniformly distributed among all sources (Figure 6.2a) and a situation where the traffic generated by each source follows a Gaussian distribution (Figure 6.2b). In addition, we report the CPU consumption for the first test (uniform distribution) in Figure 6.3.

6.5.2.1 Iptables

One of the most common approaches for DDoS attacks mitigation relies on `iptables`, a Linux tool anchored to the *netfilter* framework that can filter traffic, perform network address translation and manipulate packets. For this test we deployed all the rules containing the source IPs to drop in the `PREROUTING` *netfilter* chain, which provides higher efficiency compared to the more common `INPUT` chain, which is encountered later in the networking stack. Figure 6.2a and 6.2b show how the dropping rate of *iptables* are rather limited, around 2.5-4.5Mpps, even with a

[3]We use 64B packets because they represent the minimum size of an Ethernet frame and then are a good measure of the lower bound performance achievable by the system.



(a) (b)

Figure 6.2: Dropping rate with an increasing number of attackers. *(a)*: uniformly distributed traffic; *(b)*: traffic normally distributed among all sources.

relatively small number of attack sources, making this solution incapable of dealing with the massive DDoS attacks under consideration. This is mainly given by the linear matching algorithm used by *iptables*, whose performance degrade rapidly when an increasing number of rules are used, leading to a throughput almost equal to zero with more than 4K rules. The CPU consumption (Figure 6.3) confirms this limitation; using *iptables* to mitigate large DDoS attacks would saturate the CPUs of the system, which would be occupied discarding traffic rather then executing the target services.

6.5.2.2 Host-based mitigation

Compared to *iptables*, XDP intercepts packets at a lower level of the stack, right after the NIC driver. This test runs the entire mitigation pipeline in XDP without any help from the SmartNIC, which simply redirects all the packets to the host where the XDP program is triggered. The dropping efficiency of XDP is much higher than *iptables*, being able to discard ∼26Mpps up to 1K sources, and still ∼10Mpps with 128K attackers, using all CPU cores of the target machine[4]. This performance degradation is due to the eBPF map used (`BPF_HASH`), in which the lookup time, needed to match the IP source of the current packet against the blacklist, is influenced by the total number of map entries.

[4]In our case, the limiting factor is our Intel Xeon E3-1245 CPU, which is able to drop around 10Mpps within a single core, as opposed to other (more powerful) CPUs that are able to achieve higher rates (e.g., 24Mpps[75]).

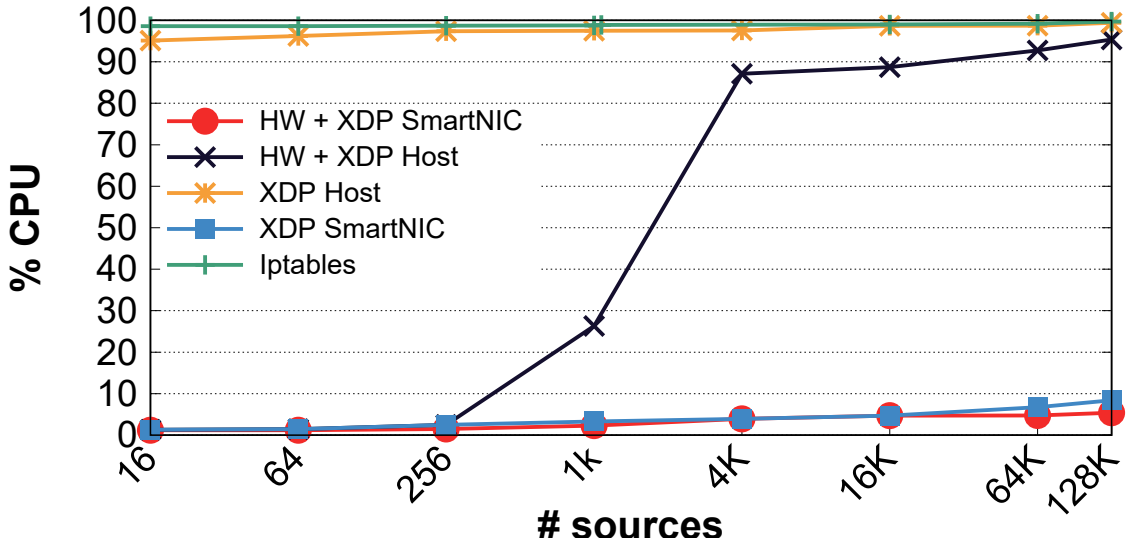


Figure 6.3: Host CPU usage of the different mitigation approaches under a simulated DDoS attack (uniform distribution).

6.5.2.3 SmartNIC-based mitigation

In this case the mitigation pipeline is executed entirely on the SmartNIC. We performed a first test where the attack is mitigated only through an XDP filtering program in the SmartNIC CPU, without any help from the hardware filter. Results shown in Figures 6.2a and 6.2b confirm a performance degradation compared to the host-based mitigation due to the slower CPU of the NIC, balanced by the fact that we do not consume any CPU cycles in the host (Figure 6.3), hence leaving room for other applications.

A second test exploits a mixture of hardware filtering and XDP-based software filtering in the card. Results demonstrate that for relatively small attack sources (less than 512), the dropping rate is equal to the maximum achievable rate (37.2Mpps); in fact, the first K rules (where K=512 in our card) are inserted in the SmartNIC hardware tables, causing all the packets to be dropped at line rate. However, when dealing with larger attacks (greater than 1K), the dropping rate immediately decreases, since an increasing number of entries stay outside the SmartNIC hardware tables; as a consequence, the dropping rate is influenced by the performance of the XDP program running in the SmartNIC CPU. This approach may be reasonable when the DDoS attack rate does not exceed the maximum achievable dropping rate in the SmartNIC CPU, which in our case is approximately 15Mpps; handling more massive attacks will cause the SmartNIC to drop packets without processing, with an higher chances to drop also legitimate traffic, as highlighted in Section 6.5.3.

6.5.2.4 Hybrid (NIC Hardware Tables + XDP Host)

In this case the offloading algorithm splits the mitigation pipeline between the SmartNIC hardware tables and the XDP *filtering* program running in the host. We notice that for large attacks, the dropping rate is considerably higher than the *HW + XDP SmartNIC* case, thanks to the higher performance of the host CPU compared to the SmartNIC one. Although hardware filtering is available also on some "traditional" NICs (e.g., Intel with Flow Director), we were unable to implement the hybrid approach in them because of the unavailability of hardware counters to measure the dropped packets for each source, which are required by our algorithm; however, we cannot exclude that other mitigation algorithms can leverage the hardware speed-up provided by the above cards as well.

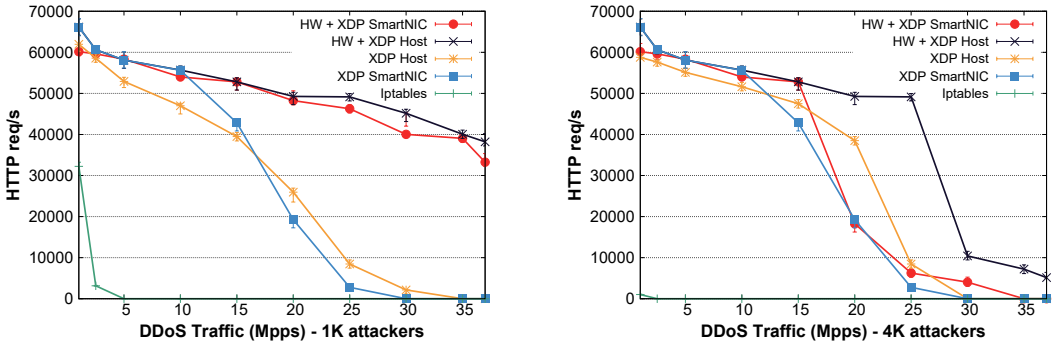
6.5.2.5 Final considerations

Figures 6.2a and 6.2b confirm a clear advantage of the hardware offloading, which is even more evident depending on the distribution of the traffic. For instance, in the second scenario (Figure 6.2b, with some sources generating more traffic than others) we can reach even higher dropping performance, thanks to the offloading algorithm that places the top-K malicious talkers in the SmartNIC, resulting in more traffic dropped in hardware. Also the CPU consumption shown in Figure 6.3 confirms the clear advantage of the offloading, particularly when most of the traffic is handled by the hardware of the SmartNIC, hence avoiding the host CPU to take care of the above portion of malicious traffic. It is worth noticing that the case where a server has to cope with a limited number of malicious sources may be rather common, as the incoming traffic in datacenters may be balanced across multiple servers (backends), each one being asked to handle a portion of the connections and, hence, also a subset of the current attackers.

6.5.3 Effect on legitimate traffic

This test evaluates the capability of the system to perform useful work (e.g., serve web pages) even in presence of a DDoS attack. We generate 64Bytes UDP packets towards the server simulating different attack rates and number of attackers, while a `weighttp` client generates 1M HTTP requests (using 200 concurrent clients) towards the `nginx` server running on the target device. The capability of the server to perform real work is reported by the number of successfully completed requests/s, with a timeout of 5 seconds, varying the rate of DDoS traffic.

Results, depicted in Figures 6.4a and 6.4b show the performance with 1K and 4K attackers respectively. In the first case, both *hardware-based* solutions reach the same number of connection/s, since almost all entries are dropped by the hardware, leaving the host's CPU free to perform real work. The same behavior can be noticed when the mitigation is performed entirely on the SmartNIC CPU; in this case, the



(a) (b)

Figure 6.4: Number of successfully completed HTTP requests/s under different load rates of a DDoS attack carried out by *(a)* 1K attackers and *(b)* 4K attackers.

host's CPU is underused, achieving the maximum number of HTTP requests/s that the DUT is able to handle. However, the performance immediately drop when the attack rate exceeds 15Mpps, which is the maximum rate that the SmartNIC CPU sustain; in such scenario, NIC queues become rapidly full, hence dropping packets without going through the mitigation pipeline and increasing the chance to drop also legitimate traffic. With respect to the *XDP Host* mitigation, we notice that the number of connections/s is initially lower, in presence of small attack rates, compared to the *SmartNIC-based* solution, since the host's CPU has to handle the HTTP requests and, at the same time, execute the XDP program. However, when the rate of the attack grows, it will continue to handle an adequate number of connections/s until 25Mpps, which is the maximum rate that the host XDP program is able to handle. Finally, *iptables*-based mitigation results unfeasible with large attack sources because of its very poor processing efficiency, severely impacting on the capability of the server to handle the legitimate traffic.

The same analysis is valid for larger attacks (e.g., 4K sources); the main difference here is that the *HW + XDP Host* solution performs significantly better in this case, thanks to the higher processing capabilities of the host's CPU compared to the SmartNIC ones.

6.6 Related work

The advantages of using XDP to filter packets at high rates have been largely discussed and demonstrated [26, 171]; several companies (e.g., Facebook, Cloudflare) have integrated XDP in their data center networks to protect end hosts from unwanted traffic, given the enormous benefits from both filtering performance and

low resource consumption. In particular, in [21] Cloudflare presented a DDoS mitigation architecture, called *L4Drop* [61] that performs packet sampling and dropping with XDP. Our approach is slightly different; we use an XDP program to extract the relevant packet headers from all the received traffic, instead of sending the entire samples to the userspace detection application and we consider simpler filtering rules, which are needed to deal with the SmartNIC hardware limitations. Finally, we consider in our architecture the use of SmartNICs to improve the packet processing, which introduces additional complexity (e.g., select rules to offload), which are not needed in a host-based solution.

6.7 Conclusions

Given the sheer increase in the amount of traffic handled by modern datacenters, SmartNICs represent a promising solution to offload part of the network processing to dedicated (and possibly more optimized) components. In this Chapter we analyzed the possibility to combine the host processing performed with eBPF/XDP, as presented in the previous chapters, with the performance speedup that may result from the hardware offloading. In particular, we explored various approaches that could be adopted to introduce SmartNICs in server-based data plane processing, assessing the achievable results in particular for the DDoS mitigation use case. We described a solution that combines SmartNICs with extended Berkeley Packet Filter (eBPF)/XDP to handle large amounts of traffic and attackers. The key aspect of our solution is the adaptive hardware offloading mechanism, which partitions the attacking sources to be filtered among SmartNIC and/or host, smartly delegating the filtering of the most aggressive DDoS sources to former.

According to our experiments, the best approach is a combination of hardware filtering on the SmartNIC and XDP software filtering on the host, which results more efficient in terms of dropping rate and CPU usage. In fact, running part of the filtering pipeline on the SmartNIC CPU would bring to inferior dropping performance due to its slower CPU, resulting in a lower capability to cope with large and massive DDoS attacks. Our findings suggest that current SmartNICs can help mitigating the network load on congested servers, but may not represent a turn-key solution. For instance, an effective SmartNIC-based solution for DDoS attacks may require the presence of a DDoS-aware load balancer that distributes incoming datacenter traffic in a way to reduce the amount of attackers landing on each server, whose number should be compatible with the size of the hardware tables of the SmartNIC. Otherwise, the solution may require the software running on the SmartNICs to cooperate with other components running on the host, reducing the effectiveness of the solution in terms of saved resources in the servers.

Chapter 7

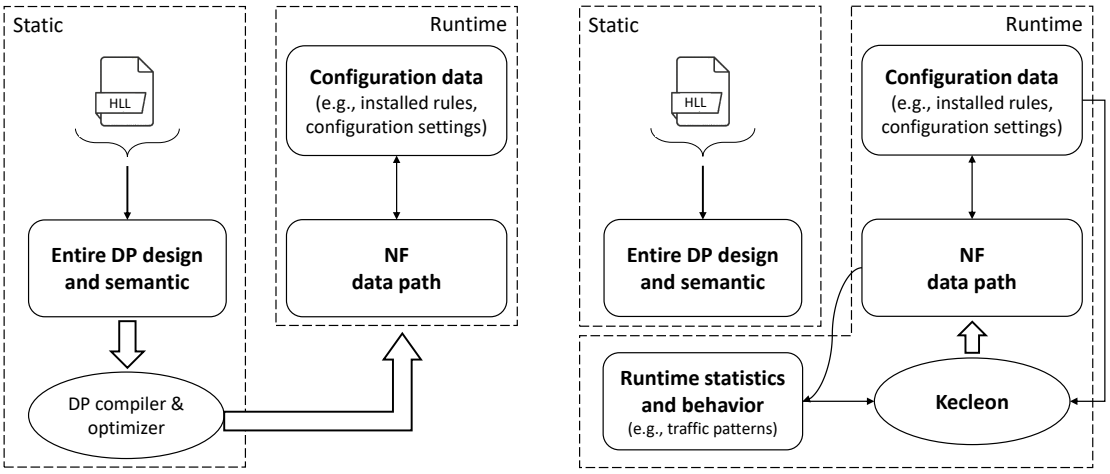
Kecleon: A Dynamic Compiler and Optimizer for Software Network Data Planes

7.1 Introduction

In recent years, the role of software packet processing in computer networking has increased significantly. Software switches, Kubernetes CNIs, service mesh data planes, software load-balancers, and software-based in-network computation are becoming increasingly popular, given the unquestionable advantages in terms of agility and rapid innovation. However, despite the improved accessibility and flexibility, the performance gap between hardware and software network data-plane is still evident. To bridge this gap and achieve a better performance-cost trade-off than the hardware, we need to optimize their performance.

In the previous chapters, we explored, thanks to the dynamic injection of eBPF programs, combined with the modular design provided by Polycube, the possibility to apply more application-specific optimizations to network functions (e.g., `bpf-iptables` in Chapter 5) at *runtime*. However, those optimizations were employed by the NF control plane, which knows the data plane behavior and that is well aware of the different configuration parameters (e.g., the composition of the fields in the ruleset) and how they are mapped into the underlying pipeline. At this point, one question arises: **Can we apply those optimizations automatically, without knowing the semantic and behavior of the NF?**

We realized that traditional approaches to design and develop those applications (both for user-space and kernel-space network functions) are based on a static compilation: the compiler receives as input a description of the forwarding plane semantic and outputs a binary code that is agnostic to the actual configuration or run-time behavior of the NF, as shown in Figure 7.1a. Since this code is executed on general-purpose servers, standard compilers (e.g., GCC, LLVM) are used to



(a) Static DP generation (b) Kecleon Dynamic DP generation

Figure 7.1: Static vs. Dynamic generation. The static compiler generates a NF agnostic data path while Kecleon takes into account runtime data, statistics, and behavior to generate the NF data path.

generate the software data-plane, which both transform the high-level code into the target machine code and apply classical compiler optimization techniques (e.g., rearrange instructions, inline functions, branch elimination) to generate a more efficient version of the original software program.

In addition to the static (offline) optimizations, which happen before the program is deployed, another branch of optimization techniques used in computer science falls under the name of *adaptive optimizations*. An adaptive optimizer can take advantage of local data conditions to *dynamically recompile* portions of the original program based on the runtime execution profile. These techniques have been successfully applied to a vast range of languages and applications, from JIT-ed languages such as JVM or JavaScript that use run-time specialization to optimize hot-code paths during the execution of the program, to database management systems to optimize the query evaluation time [168], [97]. The majority of compilers today support Profile Guided Optimizations (PGO) or Feedback Directed Optimizations (FDO) to improve the program run time performance. However, since these compilers are used to generate generic computer programs, they apply a set of optimizations and techniques that are entirely independent of the network function domain. Networking programs need a set of domain-specific optimization that are not possible without contextual information (i.e., the fact that we are processing packets, the existence of match-action-tables) that, if carefully adopted, can potentially improve the software data-plane performance [142, 141]. They often involve lots of manual tweaking and modifications (e.g., batch size, prefetching) that are not valid for every type of application, or that require a in-depth knowledge of the

application context, but that can radically change the performance [2].

In this chapter, we propose a dynamic approach to the data plane compilation where not only the static information but also the runtime data are used to generate a software data-plane, as shown in Figure 7.1b. Our dynamic compiler, **Kecleon**, uses control plane information (e.g., the type of rules installed in a firewall NFs) as input in the compilation stage to generate a custom version of the original program that is specialized (and optimized) based on the actual behavior of the NF. In addition to configuration data, Kecleon instruments the original NF data path code to retrieve the runtime packet processing behavior and statics (e.g., specific traffic patterns) that are used to optimize its output further.

As for "standard" compilers, Kecleon applies all the optimizations at the Intermediate Representation (IR) level, which make it agnostic to the language in which the NF is written and that allow us to re-use some already existing compiler optimizations at runtime. It combines static code analysis techniques to gather information about the runtime behavior of the NF and the type of operations performed inside the NF data plane. Then, it applies a set of different optimization passes (or templates) that exploit control plane information (e.g., configuration parameters), runtime data structure content, and dynamic traffic profiles to further specialize the final software data-plane.

The rest of the Chapter is organized as follows. We first motivate the need of a dynamic compiler for software data planes by taking as an example different applications, showing how the knowledge of runtime data can be beneficial for achieving higher performance (Section 7.2). We then present the main challenges that we had to address to implement a dynamic compiler for generic applications and the overall architecture of Kecleon in Section 7.3, together with a description of the different optimization passes available. Finally, we present more low-level details of the Kecleon core implementation and its eBPF plugin (Section 7.5) and a (preliminary) performance evaluation of the improvements that can be achieved with the use of Kecleon in Section 7.6. Section 7.7 concludes the Chapter.

7.2 The Case for Dynamic Network Function Optimizations

In this section, we show how the knowledge of runtime information can be exploited by a dynamic specializer to improve the throughput and latency of a given network application.

Performance depends on run-time configuration. NF software is often developed as a single monolithic block containing different features, which are usually disabled at runtime or enabled only on specific cases. Since those applications are released and used within different use cases and for various purposes, it is almost

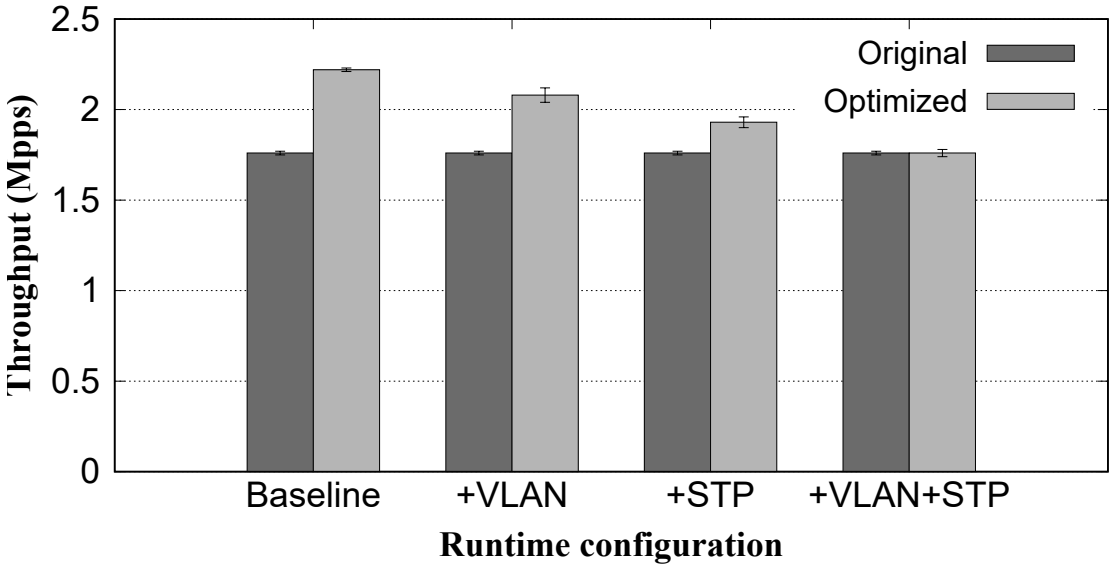
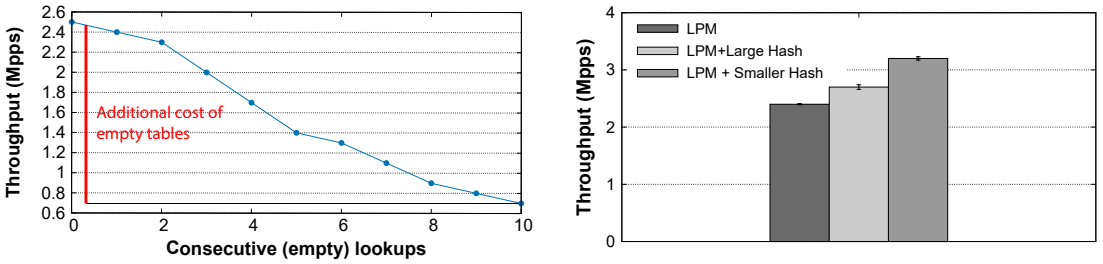


Figure 7.2: L2 bridge NF (eBPF-TC) performance with variable runtime configuration settings between the original version and the one with the unused features compiled-out; traffic is always the same across the different runtime configurations.

impossible for the vendors to distribute several versions of the same network function with different features compiled out, given the apparent difficulties that may result from their management. Moreover, although the NF developers would make a non-negligible effort to isolate the code that handles the different features, this often increases the complexity of the code, making it hard to debug and diagnose, while still leaving an additional overhead needed to check if the given configuration is enabled or not. As a result, most of the NFs today pay a non-necessary cost provided by unused features that waste additional CPU cycles and memory even if not needed at that specific moment.

For example, a L2 bridge NF developed with support for VLAN and Spanning Tree (STP) may have those features enabled only under specific circumstances or only for a subset of servers in a given cluster; however, their overhead, even when not used, may not be negligible. As we can see in Figure 7.2, the same version of an eBPF-based bridge NF, compiled with only the needed features can achieve up to 20% better performance, in terms of throughput, compared to the same version compiled with support for VLAN and STP, even if they are not used at runtime. In the same way, the run-time configuration (ruleset) of a firewall NF that supports the filtering of different fields of a packet may contain rules matching only on a subset of the entire supported fields. Even in this case, the cost paid to parse, extract, and process those additional values can be removed, saving other CPU cycles. Figure 5.8 in Chapter 5 shows that this extra cost is not indifferent.



(a) (b)

Figure 7.3: (a) Overhead given by consecutive match-action empty table lookup and (b) throughput with different map sizes and algorithms.

Takeaway: Being able to recognize and associate the configuration options installed in the control plane on a given NF with the corresponding instructions operating on those variables can allow a dynamic compiler to automatically remove those additional overheads from the NF and re-create a specialized version of the NF that is optimized for the given run-time configuration setup.

Performance depends on run-time table content. Many NFs use different data structures to store data that can be accessed when processing a packet in the pipeline. Those structures are defined by the developers when writing the NF data plane, by carefully choosing the appropriate algorithm (e.g., hash, lpm) and the size that is most appropriate for the data that they will hold. For example, a router NF may use a longest prefix match table to keep the IP addresses in the routing table or a stateful firewall NF can use a large hash table to hold the different connections that pass through it. The values in the data structures can be inserted either by an external player (e.g., the network administrator) from the control plane of the network application or by the data plane itself (e.g., conntrack table in a stateful firewall, filtering database in a L2 learning bridge).

However, at runtime, based on the current values that the table holds, the type, algorithm, or size of the data structures are not always the most efficient possible. For example, the pipeline of a vswitch may have some stages with empty ACL tables, or the routing table of a router may contain only entries with fixed IPs (e.g., /32). Of course, when designing the data plane, the developer cannot make any assumption on the type of these values, constructing it in the most general way.

Figure 7.3a shows the cost that a simple NF (in this case, the `bpf-iptables` application shown in Chapter 5) pays to perform a lookup into a set of consecutive match-action tables with no values inside; just performing a lookup to discover that the table is empty can waste a lot of CPU cycles that can be used to perform other useful tasks. At the same way, Figure 7.3b shows the throughput of a router NF that uses a LPM table to hold all the entries in the routing table; if we recognize

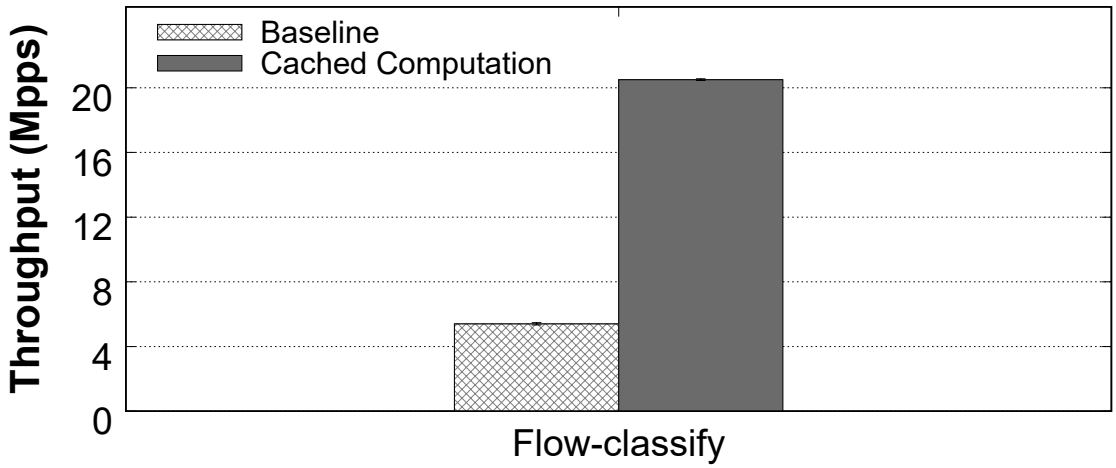


Figure 7.4: Throughput improvements when caching the computation of the top-5 flows within a high-locality trace for the DPDK *flow-classify* sample application.

that those entries can be safely installed within a more efficient table (i.e., a hash table) we can further improve its throughput. Moreover, if we use a smaller hash table as a sort of cache for the most used entries, the performance increase is even more evident since it is more likely that the small table fits entirely into a lower level cache.

Takeaway: Being able to dynamically read the content of the data structure used by the NF may allow a dynamic compiler to chose a table (or a set of tables) that is more appropriate for the current data that are stored. This is not only limited to the type of the table but also to the actual size or algorithm used.

Performance depends on run-time traffic. As for the previous cases, when developing the data plane code, the NF developer cannot make any supposition of the type of traffic the NF will receive, hence making the data plane as general as possible to perform equally independently from the traffic that is received. However, Figure 7.4 shows that, under skewed traffic patterns, we can build a fast path by partially computing values for a set of high-volume flows, drastically improving the performance of the original NF.

Takeaway: A runtime compiler that is able to transparently detect specific traffic patterns can dynamically create a fast path code that is optimized for the most common scenario, improving the performance of the original NF drastically.

7.3 Kecleon System Design

7.3.1 Design Goal

Kecleon aims at building an optimized software data path by leveraging runtime information such as the configuration of a NF or the runtime content of the match-action tables. Its main goal is to provide a unified architecture and frameworks that can be used across different software network function implementations, reusing the same type of transformations and optimizations.

To achieve this goal, Kecleon applies the optimizations at the compiler Intermediate Representation (IR) level, which in addition to the **data plane independence**, provides the following advantages:

1. *Faster compilation speed:* Once generated, Kecleon keeps the IR representation of the original program to continuously produce the optimized output data plane. All the initial compilation steps performed by the compiler's fronted (e.g., parse the program, check the syntax) do not need to be performed every time, speeding up the generation of the optimized data planes.
2. *Modularity.* Every Kecleon pass works independently from each other or uses the result of other passes as input. This would allow Kecleon to dynamically compose and chose the optimization passes that should be applied depending on the runtime data that have been obtained. Such approach also simplifies the development of new passes, which can use the results of other steps without knowing their details.
3. *Reuse existing passes.* Working at the IR level would allow Kecleon to take advantage and interact with some of the same optimizations that conventional compilers use, such as dead-code elimination, dead-store elimination, and code motion, which can be re-applied at runtime after other Kecleon transformations.

On the other hand, working at the IR level, compared to applying source-code level transformations, is way more complicated, requiring compiler's knowledge and a certain level of expertise. However, this task is required once for every new supported data plane. Kecleon defines a common interface for every plugin that is used to export the runtime information from the data plane (e.g., the content of the tables) or to apply specific transformation (e.g., create/delete a new table).

7.3.2 Design Challenges and Assumption

Challenge #1: Identify relevant logic. Developing accurate optimizations for software data-plane is challenging for a number of reasons. Many software data-planes are too generic and can be implemented in a different number of ways.

They can perform operations on network packets based on states that may be hidden deep in the data-plane code, making it extremely difficult to recognize those operations. To identify specific patterns and apply the given optimization templates we need to restrict the field of application. Although currently there is not a standard and structured approach of writing software data-plane code, the advent of programmable data planes (PDP) has increased the interest into the adoption of high-level domain-specific languages (DSL), which are specifically designed (and can be easily adopted) to express the packet processing logic of a network application. Starting from the description of the data plane that uses one of these DSL languages (e.g., P4 [33], eBPF [115, 114]), or that adopts specific framework to write NF code, Kecleon can then easily extract particular patterns into the code (e.g., code accessing to a specific structure of data or performing operations on the packets) making easier to apply the different optimizations.

For this reason, Kecleon assumes a clear distinction between stateless and stateful operations in the NF data plane. The former indicates the packet processing logic and forwarding behavior of the NF, while all the stateful operations are performed through a set of fixed functions that access to libraries of different data structures. This approach is followed by various NF frameworks (e.g., Vigor [167], Click-NF [106], BESS [73]), while other languages such as eBPF follow this approach by design (Section 2.2.1 of Chapter 2).

Challenge #2: The cost of instrumentation. Adding additional logic to characterize input traffic patterns can negatively affect performance, to the point that that it may nullify the effect of the optimization. This overhead can be reduced by instrumenting only performance-critical parts in the code.

Challenge #3: Preserve original data-plane semantic. Kecleon must ensure that all the compiler transformations and optimizations adopted do not modify the semantic of the original application. The compiler should introduce safety measures (e.g., *guards*) to restore to the original data plane code when the optimization is not valid anymore.

Challenge #4: Harmless Pipeline Swap. At each execution step, Kecleon creates a new optimized version of the original program that should replace the old pipeline. To avoid inconsistency and packet loss, this *switch* should be performed atomically, keeping unaltered the processing state in the old pipeline and preserving the same semantic of the original application.

7.3.3 Design Overview

Figure 7.5 shows the overall system architecture of Kecleon. The input of the compiler is the data plane code of the application, which is first transformed into

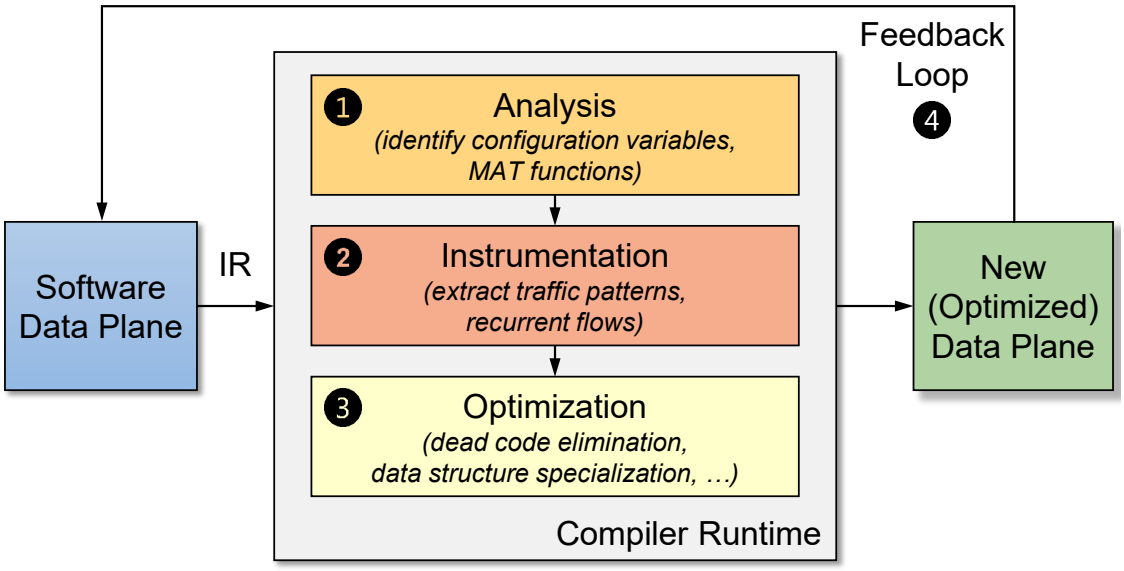


Figure 7.5: Kecleon Architecture

an Intermediate Representation (IR) - the layer where the dynamic optimization passes are applied as well as the other standard compiler optimizations.
In Kecleon, we distinguish four main steps:

1. *Analysis and logic identification* (§7.4.1): Once a new application is deployed, Kecleon starts calling the different **analysis** passes, which are in charge of analyzing the data plane code to identify specific patterns (e.g., retrieve configuration variables, operations on match-action tables). Every pass "marks" the IR code by adding some debug information to the most important instructions and stores the result of the analysis into the Kecleon internal data structures. For instance, if a given analysis pass discovers some Math-Action-Table (MAT) related methods (e.g., *lookup*, *update*), it marks the corresponding IR instruction so that the other passes do not have to perform this analysis again.

2. *Collection and retrieval of run-time statistics* (§7.4.2): After the initial analysis, Kecleon calls the set of available **instrumentation** passes, which are used to collect runtime information from running code. According to the debug info set within the previous analysis, these passes usually instruments the code by adding more instructions (e.g., to save the most frequent flows) or interact with a specific Kecleon plugin to retrieve the runtime information (e.g., read the content of a particular table).

3. *Data path optimizations* (§7.4.3): Within this phase, the actual set of **optimization** passes are called, which are in charge of performing the dynamic

transformation to the original code. They use *(i)* the result of the analysis to detect candidate methods that can be optimized and *(ii)* runtime data that are retrieved from the instrumented code or specific data structures.

4. *Pipeline substitution* (§7.4.4): Once the optimization phase is completed, Kecleon emits the optimized IR code and calls the compiler's back-end to generate the final executable. Then, it gives the control to the back-end specific plugin, which will take care of performing the actual substitution of the old pipeline with the newly generated one.

The execution of the optimization pipeline is triggered *(i)* at specific time slots or *(ii)* as a consequence of an external event. In the first case, Kecleon will repeat the entire process by checking if the runtime conditions have changed. In the second case, the pipeline can be triggered after an interaction between the control plane and the data plane, such as the update of a configuration variable or a data structure, which may invalidate the optimization applied within the previous cycle.

7.4 Kecleon Compilation Pipeline

In this section, we will go through the various steps of the Kecleon compilation pipeline. We will show how Kecleon interacts with the back-end specific plugin to identify the instructions that are used to determine the packet processing logic of the NF and, when needed, we will show more details of the implementation of the eBPF plugin for Kecleon.

7.4.1 Packet Processing Logic Identification

Identification of configuration instruction. Given their configuration agnostic implementation, most of the software data plane functions need to retrieve configuration variables at runtime from specific Match-Action-Tables (MAT) or configuration files, which can be dynamically changed by the control plane without having to restart the data plane with the new configuration. In most of the NF frameworks, this is done through the use of fixed APIs that allow both the control plane to configure those variables and the data plane to retrieve the runtime configuration. Configuration variables have the characteristic of *(i)* being generated by a single read from a configuration file or a MAT and *(ii)* never being modified during the execution of the data plane.

During the analysis phase, a Kecleon analysis pass operates as described in Algorithm 3. It scans the entire set of instructions available in the IR code and retrieves the corresponding signature, which should be identified by the back-end plugin as a configuration instruction. If the answer is positive, the method and

Algorithm 3: Identification of configuration instructions

Input: *IRM* (Module), IR of the original program
Input: *P* (Plugin), Reference to target Kecleon plugin

Function `AnalysisConfigPass`**:**

```
1  foreach IRinstr in IRM do
2      is_config ← P.IsConfigInstr(IRinstr)
3      if is_config is True and IsReadOnly(IRinstr) then
4          IRM.MarkConfig(IRinstr)
```

Algorithm 4: Identification of state instructions

Input: *IRM* (Module), IR of the original program
Input: *P* (Plugin), Reference to target Kecleon plugin

Function `AnalysisStatePass`**:**

```
1  foreach IRinstr in IRM do
2      is_table ← P.IsTableFunction(IRinstr)
3      if is_table is True then
4          tid ← P.GetTableID(IRinstr)
5          IRM.MarkTable(IRinstr)
6          IRM.StoreTID(tid)
```

the associated variable are marked, and their use is tracked along with the execution of the program. The *marking* procedure works by constructing specific debug metadata for the IR instruction, following the *DWARF* terminology[1], which can be retrieved by the other passes. Kecleon adds to the IR instruction a debug information indicating the configuration nature of the function and associates a unique *metadata* identifier that references an entry into an internal Kecleon data structure. This entry contains additional details of the method itself (e.g., if the configuration variable is stored into a MAT or not, the maximum number of possible values). The subsequent passes can then use that information to perform the optimizations.

Identification of MAT instructions. As for the configuration variables, Kecleon has to discover the instructions accessing to stateful information, which can then be retrieved and used by the subsequent optimization passes. Algorithm 4 shows the behavior of this analysis pass; it scans the entire set of instructions to find the function associated with an operation into a MAT. When found, it "marks"

[1]The Debugging With Attributed Record Formats (DWARF) is a widely recognized and standardized debug data format used to store information about a compiled computer program.

that method (e.g., the `lookup` function used to read the table content) and extracts *(i)* a unique table ID (`TID`) used to retrieve the table content at runtime, *(ii)* the layout associated with the table itself, which contains the type of data structure used (e.g., *array*, *hash*, *lpm*) and (optionally) *(iii)* a *cost function* used by Kecleon to understand the cost of performing the data structure change.

7.4.1.1 Implementation Details (eBPF Plugin)

We will now take an example the Kecleon eBPF Plugin, and we will apply the operations to a small code snippet (Listing 7.1) of the *Katran* load balancer [76], whose data plane is written in eBPF.

```
1  vip.port = pckt.flow.port16[1];
2  vip.proto = pckt.flow.proto;
3  vip_inf = bpf_map_lookup_ele(&vip_map,&vip);
4  if (!vip_inf) {
5   vip.port = 0;
6   vip_inf = bpf_map_lookup_ele(&vip_map,&vip);
7   if (!vip_inf) {
8    return XDP_PASS;
9   }
10 }
11 ...
```

Listing 7.1: Sample eBPF code from Katran NF

In an eBPF data path, the only way to set configuration values is to use specific eBPF maps, which are filled by the control plane when the program starts and retrieved at runtime from the data plane. On line 3 it performs a `lookup` into an eBPF map containing the list virtual IPs associated to a particular flow. The role of the Kecleon analysis pass is to discover that method and the associated eBPF map so that the subsequent passes can use the runtime values to optimize the new code.

Once executed, Kecleon generates the IR code of the original eBPF program, shown in Listing 7.2, which is then given as input to the different analysis passes. Following the previously-mentioned procedure, the Kecleon analysis pass will recognize the signatures at line 5 and 12 as an operation to a MAT (in this case, a *map_lookup*) and will proceed with the Algorithm 3 to check if the variable retrieved from the map is later modified in the data plane.

```
1 %226 = gep %vip, %vip* %22, i64 0, i32 1
2 store i16 %220, i16* %226, align 4
3 %227 = gep %vip, %vip* %22, i64 0, i32 2
4 store i8 %167, i8* %227, align 2
5 %228 = call i8* inttoptr (i64 1 to i8* (i8*, i8*)*)(i8* bitcast (
      @vip_map to i8*), i8* nonnull %46) #3
6 %229 = bitcast i8* %228 to %vip_meta*
7 %230 = icmp eq i8* %228, null
8 br i1 %230, label %231, label %241
```

```
 9
10  ; <label>:231:
11  store i16 0, i16* %226, align 4
12  %232 = call i8* inttoptr (i64 1 to i8* (i8*, i8*)*)(i8* bitcast (
        @vip_map to i8*), i8* nonnull %46) #3
13  %233 = bitcast i8* %232 to %vip_meta*
14  %234 = icmp eq i8* %232, null
15  br i1 %234, label %688, label %235
```

Listing 7.2: LLVM IR code of the Listing 7.1

We use the LLVM memory dependency analysis (i.e., `MemorySSA`) to reason about the interactions between various memory operations, hence detecting if the variable has been modified or not[2]. Finally, if the result of the previous analysis detects that the map and the variables are *read-only*, the Kecleon analysis pass will mark the instruction. It will allocate an internal Kecleon data structure holding the information related to that variable, such as the maximum size of the map, the layout (eBPF HASH), or the unique map identifier, which are extracted by the eBPF plugin.

7.4.2 Runtime Statistics and Data Collection

The second phase of the Kecleon compilation pipeline is composed of two steps. The former is where the actual runtime data of the previously identified instructions are retrieved; after having identified the configuration and state operations, Kecleon can retrieve all the runtime values by issuing a request to the back-end plugin. For instance, in the *Katran* example shown before, after having identified that *vip_inf* is a configuration variable, Kecleon can "ask" the eBPF plugin to retrieve the set of values within the *vip_map*.

The latter is where the original data plane code is instrumented to retrieve more specific information about the current execution of the program, which is particularly useful to adopt optimizations that improve the performance for particular traffic patterns There are several ways in which this instrumentation can be achieved. A first possible approach would be to add a local MAT into the data plane of the NF that records the values of a packet (e.g., 5-tuple) traversing the NF. However, this may incur additional and unnecessary overhead if the actions performed within the NF data plane do not use those fields. A better approach would be just to monitor the used fields, but this requires to complete an additional analysis on the NF data plane, making the instrumentation pass aware of

[2]A careful reader may argue that, if the variable is retrieved from a map with a `lookup` operation and is later modified without an `update` operation, it can still be considered as read-only. This is not valid for eBPF, where the result of a lookup is a *pointer* to the entry in the map; as a consequence, a modification on the pointer's content will modify the corresponding map.

the actions performed on the packets or depending on its content.

To avoid this, Kecleon adopts an **implicit traffic-specific** mechanism by automatically instrumenting the sections of the code that access or modify the internal state of the NF. In particular, the Kecleon instrumentation pass retrieves all the MAT accesses and adds a Local MAT that stores the same data of the original table. It samples only the most-accessed entries within the original MAT and their corresponding values, saving only a limited number of entries to reduce its size and overhead. Then, the Kecleon platform-specific plugin exports the entries from the instrumented MAT using the Kecleon common data format so that the various optimization passes can easily consume them in a target-independent way.

This approach brings several advantages. First, it does not require any static analysis on the NF code to retrieve a common set of packet values that should be representative of the incoming traffic. Second, it gives to Kecleon a more fine-grained control of instrumentation that is applied. If it recognizes that the overhead of a single table is minimal or there is no space for improvements, the instrumentation can be disabled only for that specific branch or table, while still allowing to retrieve information on the other methods. Kecleon can change at runtime the size of the instrumented tables according to the level of information it needs and the sampling rate of the instrumented entries, which is a compromise between accuracy of the instrumented traffic and performance overhead introduced by the instrumentation. Finally, merging all the information coming from the different local MAT, it is possible to reconstruct the hot code paths and then optimize for them.

7.4.2.1 Implementation Details (eBPF Plugin)

We show here a small example of the `bpf-iptables` network function presented in Chapter 5. In Listing 7.3 we illustrate a short code snippet of module matching the L4 ports of the packet, excluding the colored code that we will explain later. On line 8, a *lookup* in the *port_map* is performed to retrieve the bitvector associated; then from line 11 to 15 the final bitvector is calculated by performing a bitwise *and* operation with the current bitvector obtained in the other steps of the pipeline. If the resulting bitvector is zero, the default action is applied (e.g., the packet is dropped); otherwise, the next module in the pipeline is called.

```
1  u64 *val = bpf_lookup(&port_map_c, &l4port);
2  (*val)++;
3  bpf_map_update(&cpu_port_map, &l4port, val);
4  if (l4port == 8080) {
5      isAllZero = True;
6      goto NEXT;
7  }
8  ele = bpf_lookup(&port_map, &l4port);
9  ...
10 #pragma unroll
11 for (i = 0; i < _NR_ELEMENTS; ++i) {
```

```
12    bits[i] = bits[i] & (ele->bits)[i];
13    if (res->bits[i] != 0)
14      isAllZero = false;
15  }
16  goto NEXT;
17  ...
18  NEXT:;
19  if (isAllZero) {
20      applyDefaultAction();
21      return;
22  }
23  call_bpf_program(ctx, _NEXT_HOP_1);
24  ...
```

Listing 7.3: Sample code of the bpf-iptables L4-Port matching module

In the eBPF case, the Kecleon instrumentation uses a *per CPU* `BPF_HASH` map to store all the L4Ports that reaches the original map with a corresponding counter[3]. The *red* code in Listing 7.3 shows the corresponding C code that would be added by the instrumentation pass.

The Kecleon optimization pass can then try to optimize the code using the result of the instrumentation. For instance, in this example, if most of the runtime flows contains packets with L4 port 8080, and the associated bitvector[4] contains all bits at zero, a Kecleon optimization pass can pre-compute the value (*green* code of Listing 7.3), and the loop from line 11 to 15 will be avoided, saving a lot of CPU cycles for the most common case.

7.4.3 Kecleon Data Path Optimizations

In this section, we will describe of the main Kecleon optimization passes and how they exploit the runtime information gathered during the previous steps to generate an optimized data plane.

7.4.3.1 Dead Code Elimination Pass (DCE)

The goal of the Kecleon dynamic DCE pass is equivalent to the corresponding DCE pass used in most of the compiler optimization phases; it prunes "dead" branches and instructions that are considered unreachable at compilation time.

[3]The instrumented tables can also have entries that are not contained in the original map; this information is also beneficial to the optimization passes that may optimize the code for the most common case.

[4]Kecleon does not know that the map value is a "bitvector", it just substitutes value in the code and tries to find a better path using the already existing compiler (LLVM) optimization passes.

Algorithm 5: Kecleon Dynamic DCE algorithm

Input: *IRM* (Module), IR of the original program
Input: *P* (Plugin), Reference to target Kecleon plugin

Function `DynDCEPass`**:**

```
1  forall IRConfInstr in IRM do
2      runtime_val ← P.GetConfigValue(IRConfInstr)
3      if runtime_val.size() > 1 then
4          foreach val in runtime_val do
5              P.IRBuilder.PropagateValueSet(val, IRConfInstr)
       else
6          P.IRBuilder.PropagateSingleValue (runtime_val,
             IRConfInstr)
7      RunConstantPropagationPass(IRM)
8      RunConstantFoldingPass(IRM)
9      RunDCEPass(IRM)
10 return IRM
```

This pass works as described in Algorithm 5. As the first step (line 2), the algorithm scans the entire set of IR instructions to find the one marked as *configuration* (as described in Section 7.4.1). Then, it interrogates the back-end plugin to retrieve the runtime values for the associated configuration variable. It is important to note that the exported variables can assume different values and types; it is the role of the Kecleon back-end plugin to provide the required information according to the defined plugin interface. For example, if a configuration variable is a list, a parameter `is_list` is stored into the internal Kecleon data structures during the analysis step; then, the various optimization passes can use this information to act differently based on their goal. As the second step (line 3-6), the algorithm substitutes the original configuration instruction with the runtime value (or set of values). Then, it can directly call the existing compiler passes to automatically perform *Constant Propagation*, *Constant Folding*, and *Dead Code Elimination* (line 7-9) that will eliminate the unreachable code.

The DCE Pass mentioned above can remove redundancy in the NF code, but there are some scenarios where it cannot be fully applied. It may happen that, after the propagation of the configuration, only a subset of the paths into the code is subjected to the dead code elimination. Within the current state, Kecleon is not able to recognize this scenario, given the path-insensitive type of analysis that is applied. A solution would be to use symbolic execution methods to perform a path-aware DCE, then merge the results of each path to generate the final output. Although not currently implemented, this extension is part of the future work.

Algorithm 6: Kecleon Dynamic DSS algorithm

Input: IRM (Module), IR of the original program
Input: P (Plugin), Reference to target Kecleon plugin

Function `DynDSSPass`**:**

```
1  forall IRTableInstr in IRM do
2      tid ← IRM.RetrieveTID(IRTableInstr)
3      info ← P.GetTableInfo(tid)
4      values ← P.GetTableValues(tid)
5      if IsDSSFeasible(info, values) then
6          new_info = ApplyDSSToTable(info)
7          P.CheckCostFunction(new_info, info)
8          P.IRBuilder.CreateNewTable(new_info, IRTableInstr)
9  return IRM
```

7.4.3.2 Data Structure Specialization Pass (DSS)

The Kecleon DSS Pass is in charge of analyzing the runtime content of the MATs used by the NF data plane and modify their layout, size, or algorithm to the one that better performs under the given runtime conditions.

For example, knowing which field of a key in a Longest Prefix Match (LPM) table indicates the prefix, under specific circumstances (e.g., all entries have the same prefix) the Kecleon DSS could automatically convert the LPM table into a hash table. Algorithm 6 shows the behavior of this pass.

It starts by extracting the runtime values from the map using the Table Identifier (TID) taken by the previous analysis pass and retrieved from the debug information associated with the IR instruction. Then, depending on the runtime values, it decides if the data structure content matches a possible transformation. If a *cost function* is given, Kecleon first checks if the new change may provide the expected performance benefits; otherwise, the transformation is discarded. For frameworks that offer a clear definition of the data structures and their implementations, their associated cost function can be automatically deducted using static analysis or symbolic execution methods [124], [126], instead of requiring a manual effort from the plugin developer.

Note: Today, many NFs are stateful, where packet processing updates states that, in turn, influences the packet data path. Changing the data structure used by the data plane of a NF may be dangerous if the table is modified within the data plane itself. If the newly inserted entries invalidate the assumption used by the DSS Pass when it has performed the optimization, we may corrupt the original application semantic. As a consequence, Kecleon applies the DSS Pass only to tables that are not modified in the data plane (the DSS Analysis Pass can recognize this situation). A change in those tables can happen only from the control plane, and, in this

scenario, Kecleon triggers the execution of the optimization pipeline to re-evaluate the feasibility of the optimization.

7.4.3.3 Cached Computation (CC) Pass

The CC Pass can be seen as a further specialization of the DSS pass, where together with the runtime MAT values also the results of the instrumented MAT are used. The results of the instrumentation give a hint to the different optimization passes about the most common paths into the code or the most used entries in a MAT. The Cached Computation (CC) Pass can reduce the overhead given by specific memory accesses by, as the name suggests, caching the computation of the most accessed entries within the code itself or in other, more efficient, tables. For example, a simple `lookup` operation in a large MAT may be in charge of 70% of the overall overhead of the NF, causing a lot of cache misses, even if the number of matched entries (taken from the runtime instrumentation) represent only the 5% of the overall number of entries in the table. The CC Pass can take the 5% of the most used entries within the MAT table and compile them directly in the code, by pre-computing the result of the MAT lookup for those entries. A parameter controls the maximum number of entries under which a table is directly compiled. The pre-computation depends on the original layout of the compiled table and the type of variable used for the lookup. For example, the cached entries within a *hash* table are converted into a series of *switch-case* matching the single key value used in the lookup, or a hash of this value for complex entries[5]. In the latter case, a pre-computed collision-free hash (among the other cached values) is used to access to the pre-computed variable. In the same way, for an `LPM` table of IP addresses, we could derive from the instrumentation a set of most-accessed IPs, whose lookup result can be pre-computed and cached within the code, using the same procedure described before.

Avoid inconsistency of cached entries. When the Kecleon CC Pass caches some entries directly in the code it needs to ensure that a subsequent modification in the table, which may happen both from the control plane and the data plane for non-configuration MAT, is reflected immediately in the cached code that should then use the updated information instead of the old (directly compiled in the code) one. That means that packets coming after the update should be immediately redirected to the original code branch (i.e., performing regular table lookup) instead of accessing the cached information. Triggering the execution of the Kecleon optimization pipeline every time an update is found would result on new packets coming

[5]With complex entries we indicate variables that contains more than one primitive value (e.g., a C `struct`).

immediately after the update to match the old cached version until Kecleon emits a new pipeline, breaking one of the main Kecleon assumption to keep the original data plane semantic untouched.

To avoid this issue, Kecleon makes use of **guards**. A *guard* is a control variable (or a specific MAT) that contains the version of the currently cached computation that is compiled in the code; before accessing the CC, the NF data plane checks if the value contained in the guard matches the one of the compiled version - if not, it falls back to the "default" path, which corresponds to the original NF data plane. The update of the guard is done atomically before the map associated with it is modified (of course, if the update does not impact the cached entries, the guard is not modified), and the way it is done depends on the given target platform.

For example, in the eBPF plugin implementation, the *guard* map is implemented as an additional `PER_CPU_ARRAY` table with a single entry containing the current version of the code. Then, an additional *kprobe* is attached to the `bpf_map_update` helper; therefore, when an update is performed, the Kecleon eBPF program attached to the kprobe is executed and it updates the guard consequently, guaranteeing the consistency of the original data plane. Of course, the use of guards increases the overhead of the NF since it wastes some CPU cycles to check its version before accessing the cached entries. We have measured its cost in Section 7.6.

7.4.4 Kecleon Pipeline Update

The execution of the Kecleon optimization pipeline can be triggered *(i)* as a consequence of an intercepted control plane event (e.g., the update of a configuration MAT) or *(ii)* periodically at given time slots. Kecleon performs the updates of the original data plane in a non-destructively way. When all the optimizations are applied, Kecleon calls the rest of the compiler pipeline to convert the IR representation of the NF data plane into the target native language where the data plane is executed. During the rebuild, the old running data-path can continue processing packets without any service disruption. The newly created data-path (within its binary form) is then passed to the specific Kecleon plugin that has the role of injecting and substituting it with the previous pipeline.

To do so, Kecleon installs, before the original pipeline, an additional program whose purpose is to "jump" to the first instruction of the pipeline, using the address of the new code, through an *indirect jump* (or *trampoline*). Basically, when the new code is ready, it is loaded (and properly relocated) to a new address into the process address space that will be pointed by the *trampoline* instruction. The new pipeline representation is constructed side by side with the running data-path, it is then inserted into the pipeline by atomically redirecting this *jump* instruction to the address of the new code.

Note: As we have seen in the previous chapters, Polycube provides a more natural way to perform this pipeline substitution, thanks to the use of the *program array*

map, whose role is to hold a set of "pointers" to different eBPF programs. By performing a *tail-call* to a given index of the array map, the corresponding eBPF program is called. In our case, by updating the index of the old pipeline with the new one we can quickly achieve the *swap* behavior.

7.5 Prototype Implementation

Kecleon is a specialization of an existing compiler that adds networking domain information to existing (or new) compiler optimizations passes by extracting the needed information from the source code and the runtime execution of the NF, a similar mechanism used by Just-In-Time (JIT) compilers.

Kecleon Core. We implemented Kecleon using the LLVM [99] framework (v10.0.0) and, in particular, through the *MCJIT* execution engine, which provides an environment for the code generation and the manipulation of the original data plane code within its IR representation. Kecleon is a separate daemon running in the system that receives the NF code. It converts the code into the LLVM IR representation (i.e., a single or a set of LLVM *Module* objects), which are then analyzed and optimized through the various Kecleon optimization passes. The conversion between the source code and the LLVM IR representation is done by the compiler front-end (e.g., *Clang* for languages in the C language family); Kecleon does not perform any manipulation on the Abstract Syntax Tree (AST) of the compiler front-end and then is independent from the front-end. It would then be possible to run Kecleon by directly providing the IR representation of the original program (i.e., the `.bc` file) instead of the entire source code.

7.5.1 eBPF Plugin

The Kecleon eBPF plugin is implemented as part of the BPF Compiler Collection (BCC), which already includes a set of abstractions and helper functions to interact with the eBPF APIs. Also, BCC is already built around the Clang/LLVM toolchain, hence simplifying the interaction between the eBPF source code and the rest of the compiler toolchain. Since Polycube (§4) is also builts around BCC, this Kecleon eBPF plugin implementation can be used to dynamically (and automatically) optimize Polycube NFs, as we will see in Section 7.6.

7.6 Evaluation

The goal of this evaluation section, although still in a preliminary phase, aims at validating *(i)* the performance improvements that real-world applications can

achieve by applying the Kecleon dynamic compilation framework and *(ii)* how much operational and performance Kecleon can introduce in the normal NF workflow.

7.6.1 Setup

Our testbed includes a first server used as DUT running the firewall under test and a second used as packet generator (and possibly receiver). The DUT encompasses an Intel Xeon Gold 5120 14-cores CPU @2.20GHz (hyper-threading disabled) with support for Intel's Data Direct I/O (DDIO) [84], 19.25 MB of L3 cache and two 32GB RAM modules. The packet generator is equipped with an Intel® Xeon CPU E3-1245 v5 4-cores CPU @3.50GHz (8 cores with hyper-threading), 8MB of L3 cache and two 16GB RAM modules. Both servers run Ubuntu 18.04.1 LTS, with the packet generator using kernel 4.15.0-36 and the DUT running kernel 4.19.0. Each server has a dual-port Intel XL710 40Gbps NIC, each port directly connected to the corresponding one of the other server.

7.6.2 eBPF NFs (Polycube)

In this section, we took some of the eBPF-based NFs implemented in Polycube, and we evaluated the corresponding performance improvements that result from the use of Kecleon to optimize their pipeline. Figure 7.6 shows the result of this evaluation for four different NFs.

The effectiveness and performance advantages brought by Kecleon depend on different factors. The first one, is the way the data plane of the NF is written. We haven't performed any modification to the original code of the Polycube NFs under consideration but, it is important to note that most of these NFs are developed with the dynamic reloading idea available within eBPF and Polycube. Therefore, there are a lot of "manual" modification already applied to the source code that further improves the speed of the original NF data plane. Nevertheless, the performance advantages brought by the use of Kecleon are evident with an improvement range that goes from 20 to 50% under different scenarios.

The other two factors that affect the performance and the evaluation results are the runtime configuration of the NF and the input traffic that is received. Kecleon uses instrumentation to gather runtime information about the traffic received by the NF and optimize the data path accordingly. If the traffic has a well-defined behavior and a high locality, the improvements with the use of Kecleon are more evident. The same is valid for the type of MAT values and configuration setting. We have evaluated ten different configuration settings and 20 various input traffic traces for each NF under consideration; we have reported the *average*, *maximum*, and *minimum* throughput obtained under the different runs in Figure 7.6. We will further investigate the impact of the configuration and input traffic in Section 7.6.3.

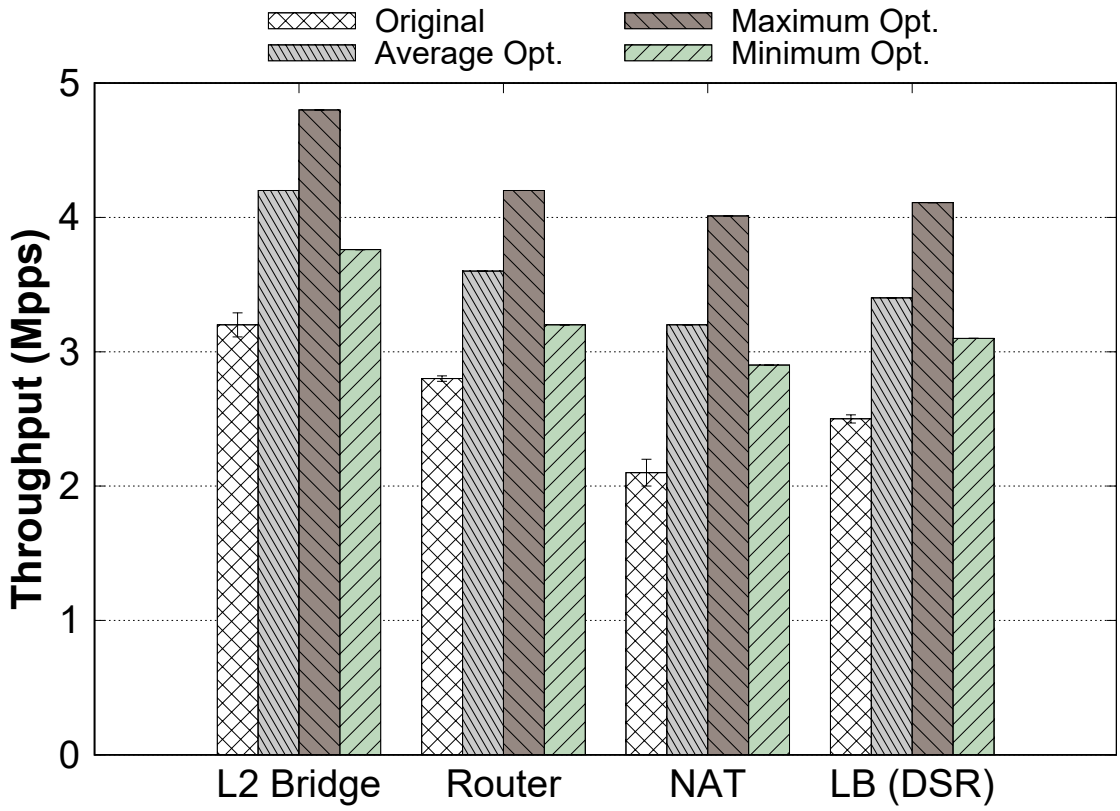


Figure 7.6: Single-core throughput for various Polycube eBPF-based NFs. We report the average, maximum, and minimum throughput values under different configuration and traffic setup when using Kecleon to generate the optimized code.

7.6.3 eBPF-firewall (`bpf-iptables`)

In this section, we evaluated the effectiveness of Kecleon with a more complex application, such as the one presented in Chapter 5, i.e., `bpf-iptables`. We have used Classbench [155] to generate ten different configuration settings for the firewall and tested them with three various packet traces with different localities of traffic. The *no locality* trace contains traffic that is uniformly distributed among the entire ruleset; this is the worst scenario for Kecleon optimization passes that rely on the results of the instrumentation to optimize the output code (e.g., the CC Pass in Section 7.4.3.3). On the other hand, the *high locality* trace contains a set of elephant flows matching a subset of the entire ruleset, which is a more favorable scenario for the Kecleon dynamic passes.

As we can see in Figure 7.7, in all three different cases, the Kecleon dynamic compilation approach provides performance advantages, which are more evident when a high locality trace is used. With the *no locality* trace, the improvement is less visible since the cost paid for the instrumentation impacts more than the

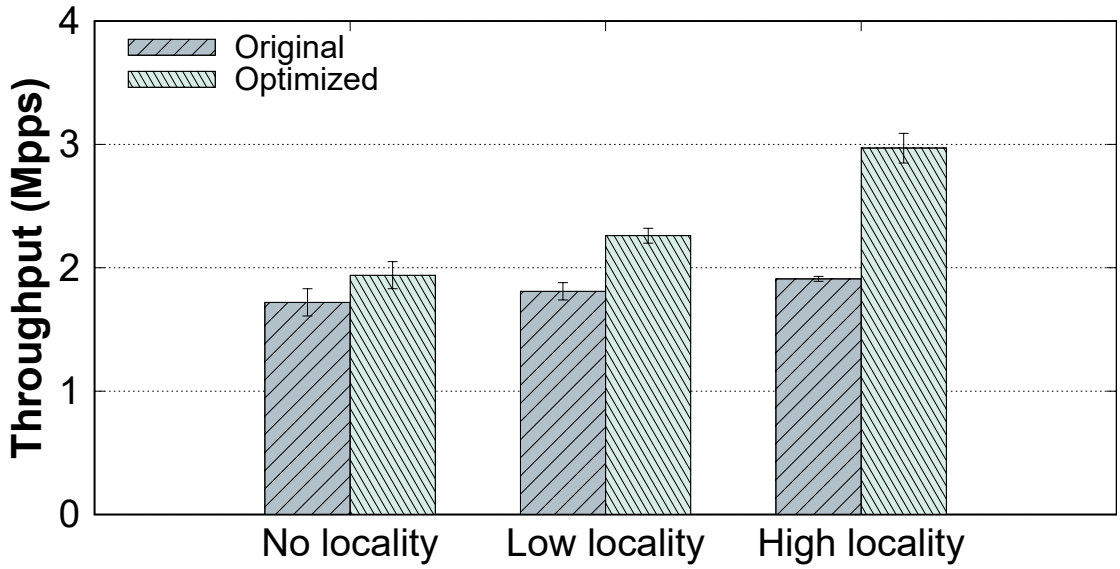


Figure 7.7: Single-core throughput for `bpf-iptables`. We report the throughput under three different Classbench generated traces with no locality (traffic is uniformly distributed) to high locality (few elephant flows).

enhancement that we can get from the instrumentation-based optimization passes and, some time can even invalidate the improvements of the other passes (e.g., DSS or DCE). We will evaluate the cost of the instrumentation in Section 7.6.4.1.

7.6.4 Microbenchmarks

7.6.4.1 Instrumentation overhead

The overhead of Kecleon comes mainly from the instrumentation phase, where the original data plane code is "augmented" with additional instructions used to gather runtime data from the running data path, which is done automatically by Kecleon. In Figure 7.8 we evaluated the cost of this overhead for the `bpf-iptables` application. The results showed are taken from ten different runs with ten different configuration settings and traffic profiles, with the average result displayed in the figure. We disabled for this test the optimization passes so that only the overhead is calculated. As we can see, the cost of the instrumentation is non-negligible, and it justifies the "low" improvement with the no locality trace shown in Figure 7.7. Although better approaches exist in literature to achieve the same goal [78, 146, 164] with a lower cost and better efficiency (e.g., count–min sketches), they require the use of special data structures or operations that may not be available in every target support. On the other hand, our approach to applying local MATs, although more costly, does not require special requirements in terms of data structures.

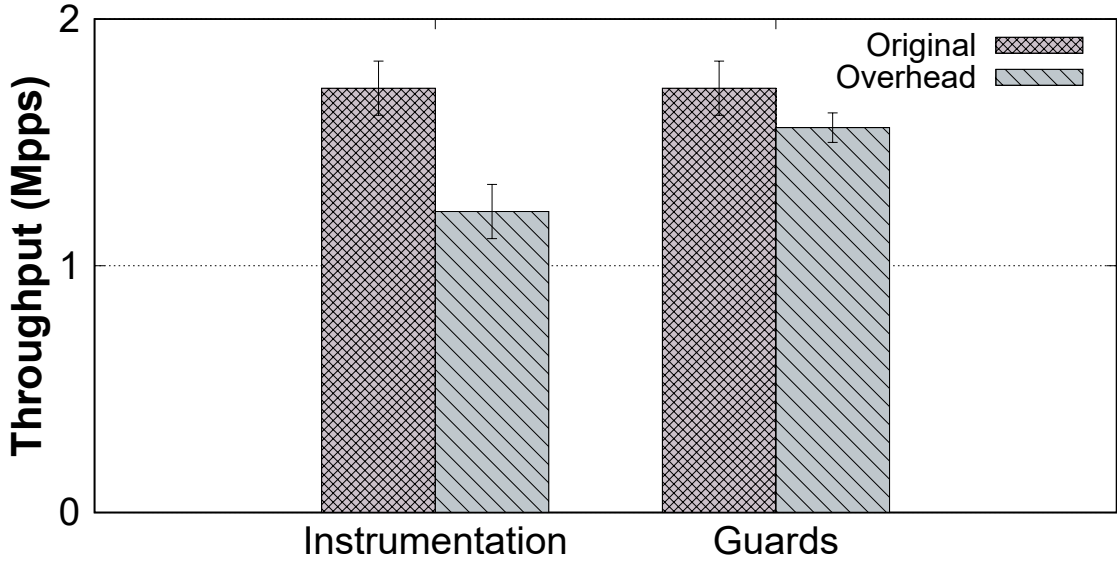


Figure 7.8: Single-core throughput for `bpf-iptables` when only the Kecleon instrumentation is applied, without any of the optimization passes to take effect. The first bar indicates the overhead of the implicit traffic-specific approach used by Kecleon, while the second indicates the overhead for the use of *guards* tables.

Finally, we have also measured the overhead of the *guards* table, which are used by the Kecleon CC pass to guarantee the original NF semantic when the various transformations are applied. The cost of these tables is less evident because differently from the instrumentation tables, they are used only when a stateful operation is found; in `bpf-iptables` only one map, the *conntrack* table, is used to save the actual state of the connections.

7.7 Conclusions and Future Works

In this Chapter, we have first demonstrated that the performance of most of the software data planes today are heavily influenced by runtime parameters, which cannot be exploited with the standard "static" compilation process. We presented Kecleon, a runtime compiler for software data planes that exploits runtime information, such as configuration data, runtime table contents or current execution profiles, to generate an optimized version of the original data plane that is more optimized for its runtime behavior.

Kecleon compilation pipeline is divided into three different steps. An analysis phase that is used to identify and extract the packet processing logic of the application. This step works with the assumption that all the data plane code under analysis is written according to a DSL language that provides common abstraction

and a fixed set of APIs to perform stateful operations. Then, an instrumentation phase is used to monitor and extract the current execution profiles of the data plane. Finally, an optimization phase uses all the previously obtained data to optimize the code. For example, it removes redundant instructions that are not hit within the current configuration and execution profiles, or it substitutes a MAT with a more efficient version based on its content.

We have evaluated the performance of Kecleon for different eBPF-based NFs and we have noticed how it is effective under several configuration settings and traffic profiles.

Future works. We have implemented in Kecleon all the previously mentioned compiler passes and the eBPF plugin; however, we are still missing the evaluation of other real-world applications such as OvS-eBPF, Cilium, and Katran, which are developed independently. We are planning to include a DPDK plugin for Kecleon so that it can be used within a vast range of applications and DPDK-based frameworks. According to our "manual" experiments, the performance benefits that we can get from those applications are even more evident, given the greater control of operations that can be performed and, consequently, of the optimization that Kecleon can apply.

Finally, as mentioned in Section 7.4.3.1, we are planning to extend the type of analysis that is currently performed in Kecleon by adding a path-aware analysis. Through symbolic execution and program slicing methods, Kecleon can apply even more aggressive optimization that are not possible within the current prototype.

Chapter 8

Concluding Remarks

In this dissertation, we explored the new challenges and requirements that software packet processing is facing in the microservice era. Traditional monolithic approaches to build and run virtual network services result contradictory to the cloud-native paradigm, and simply transforming those monolithic patterns into analogous cloud-native counterparts results in manageability, scalability, and performance problems. Our attempt to solve this issue brought us to the design and implementation of Polycube and Kecleon, and to an additional study of how SmartNICs can be used to improve such network services. The former's goal is to provide a common infrastructure to build, manage, and run such networking services that follow the same paradigms on microservices applications. Polycube services can be dynamically combined to form in-kernel service chains and can be externally controlled through a specific set of service-specific APIs that allow customizing the service behavior. The Polycube service data plane, which uses the extended Berkeley Packet Filter (eBPF), enables an unprecedented level of customization, programmability, and performance improvements that were not possible before while keeping compatibility with existing applications and with "native" environments. On the other hand, Kecleon enables an automatic specialization of those network services by exploiting the runtime knowledge of configuration data, data structures content and execution profiles to automatically generate an optimized version of the original network function data plane that is more efficient for current runtime behavior.

In the future, we envision a novel model of software network functions that satisfy the following conditions. First, they should be able to provide advanced, programmable and efficient services to cloud-native applications, while being able to dynamically adapt to the continuously changing run-time environment and configuration conditions that are common within this scenario. Second, they should adhere to the same design patterns and concepts of the microservice paradigms, enabling a continuous delivery workflow, horizontal scaling while efficiently using network resources and greater control through well-defined APIs.

We hope that the combination of the works presented in this dissertation can lay the foundation for a new model of packet processing applications that can be dynamically re-combined, re-generated and re-optimized without sacrificing programmability, extensibility and performance. We release the source code of all the systems presented in this thesis and we hope they will be useful for both researchers and industrial system developers to build upon them.

Appendix A

List of Publications

The following is the complete list of publications carried out during the Ph.D.

- **Miano, S.**, & Risso, F. (2020, March). A Micro-service Approach for Cloud-Native Network Services. In 2020 *ACM SIGCOMM Symposium on SDN Research (SOSR)* ACM.
- **Miano, S.**, & Risso, F. (2020). Transforming a traditional home gateway into a hardware-accelerated SDN switch. *International Journal of Electrical and Computer Engineering*, 10(3), 2668.
- **Miano, S.**, Bertrone, M., Risso, F., Bernal, M. V., Lu, Y., Pi, J., & Shaikh, A. (2019, September). A Service-Agnostic Software Framework for Fast and Efficient In-Kernel Network Services. In 2019 *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)* (pp. 1-9). IEEE.
- **Miano, S.**, Doriguzzi-Corin, R., Risso, F., Siracusa, D., & Sommese, R. (2019). Introducing SmartNICs in Server-Based Data Plane Processing: The DDoS Mitigation Use Case. *IEEE Access*, 7, 107161-107170.
- **Miano, S.**, Bertrone, M., Risso, F., Bernal, M. V., Lu, Y., & Pi, J. (2019). Securing Linux with a faster and scalable iptables. *ACM SIGCOMM Computer Communication Review*, 49(3), 2-17.
- **Miano, S.**, Bertrone, M., Risso, F., & Tumolo, M. (2018, August). Accelerating linux security with ebpf iptables. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos* (pp. 108-110).
- Bertrone, M., **Miano, S.**, Pi, J., Risso, F., & Tumolo, M. (2018). Toward an eBPF-based clone of iptables. *Netdev'18*.

- **Miano, S.**, Bertrone, M., Risso, F., Tumolo, M., & Bernal, M. V. (2018, June). Creating complex network services with ebpf: Experience and lessons learned. In 2018 *IEEE 19th International Conference on High Performance Switching and Routing (HPSR)* (pp. 1-8). IEEE.

- **Miano, S.**, Risso, F., & Woesner, H. (2017, July). Partial offloading of OpenFlow rules on a traditional hardware switch ASIC. In 2017 *IEEE Conference on Network Softwarization (NetSoft)* (pp. 1-9). IEEE.

- Bonafiglia, R., **Miano, S.**, Nuccio, S., Risso, F., & Sapio, A. (2016, October). Enabling NFV services on resource-constrained CPEs. In 2016 *5th IEEE International Conference on Cloud Networking (Cloudnet)* (pp. 83-88). IEEE.

Bibliography

[1] Zaafar Ahmed, Muhammad Hamad Alizai, and Affan A. Syed. "InKeV: In-Kernel Distributed Network Virtualization for DCN". In: *SIGCOMM Comput. Commun. Rev.* 46.3 (July 2018). ISSN: 0146-4833. DOI: 10.1145/3243157.3243161. URL: https://doi.org/10.1145/3243157.3243161.

[2] Omid Alipourfard and Minlan Yu. "Decoupling Algorithms and Optimizations in Network Functions". In: *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*. 2018, pp. 71–77.

[3] Mohammad Alizadeh et al. "CONGA: Distributed Congestion-Aware Load Balancing for Datacenters". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM '14. Chicago, Illinois, USA: Association for Computing Machinery, 2014, pp. 503–514. ISBN: 9781450328364. DOI: 10.1145/2619239.2626316. URL: https://doi.org/10.1145/2619239.2626316.

[4] Esraa Alomari et al. "Botnet-based distributed denial of service (DDoS) attacks on web servers: classification and art". In: *arXiv preprint arXiv:1208.0403* (2012).

[5] Arbor Networks. *Worldwide Infrastructure Security Report*. 2017. URL: https://pages.arbornetworks.com/rs/082-KNA-087/images/13th%5C_Worldwide%5C_Infrastructure%5C_Security%5C_Report.pdf (visited on 03/17/2019).

[6] AT&T. *AT&T DIRECTV*. [Online; last-retrieved 08-April-2020]. 2020. URL: https://www.att.com/bundles/directv-wireless/.

[7] BCC Authors. *HTTP Filter*. [Online; last-retrieved 15-November-2018]. Feb. 2016. URL: https://github.com/iovisor/bcc/tree/master/examples/networking/http_filter.

[8] Cilium Authors. *BPF and XDP Reference Guide*. July 2018. URL: https://cilium.readthedocs.io/en/latest/bpf/ (visited on 03/17/2019).

[9] Cilium authors. *Diagram of Kubernetes / kube-proxy iptables rules architecture*. Jan. 2019. URL: https://web.archive.org/web/20200414131802/https://github.com/cilium/k8s-iptables-diagram.

[10] Lighttpd authors. *weighttp: a lightweight and simple webserver benchmarking tool*. [Online; last-retrieved 10-November-2018]. Nov. 2018. URL: https://web.archive.org/web/20190718231322/http://redmine.lighttpd.net/projects/weighttp/wiki.

[11] Netfilter Authors. *Moving from iptables to nftables*. [Online; last-retrieved 10-October-2018]. Oct. 2018. URL: https://web.archive.org/web/20200308125421/https://wiki.nftables.org/wiki-nftables/index.php/Moving_from_iptables_to_nftables.

[12] Polycube Authors. *Polycube: eBPF/XDP-based software framework for fast network services running in the Linux kernel*. [Online; last-retrieved 22-July-2019]. Jan. 2019. URL: https://github.com/polycube-network/polycube.

[13] The Network Service Mesh authors. *What is Network Service Mesh?* Jan. 2020. URL: https://web.archive.org/web/20200414131411/https://networkservicemesh.io/docs/concepts/what-is-nsm/.

[14] Pablo Neira Ayuso. *[PATCH RFC PoC 0/3] nftables meets bpf*. [Online; last-retrieved 29-March-2019]. Feb. 2018. URL: https://web.archive.org/web/20191024111400/https://www.mail-archive.com/netdev@vger.kernel.org/msg217425.html.

[15] Paul Baran. "On distributed communications networks". In: *IEEE transactions on Communications Systems* 12.1 (1964), pp. 1–9.

[16] Tom Barbette et al. "A High-Speed Load-Balancer Design with Guaranteed Per-Connection-Consistency". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 667–683. ISBN: 978-1-939133-13-7. URL: https://www.usenix.org/conference/nsdi20/presentation/barbette.

[17] BCC authors. *BPF Compiler Collection (BCC)*. URL: https://web.archive.org/web/20181106133143/https://www.iovisor.org/technology/bcc.

[18] Sunny Behal and Krishan Kumar. "Detection of DDoS attacks and flash events using information theory metrics–an empirical investigation". In: *Computer Communications* 103 (2017), pp. 18–28.

[19] Sunny Behal and Krishan Kumar. "Detection of DDoS attacks and flash events using novel information theory metrics". In: *Computer Networks* 116 (2017), pp. 96–110.

[20] Mauricio Vásquez Bernal et al. "A Transparent Highway for Inter-Virtual Network Function Communication with Open VSwitch". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brazil: Association for Computing Machinery, 2016, pp. 603–604. ISBN: 9781450341936. DOI: 10.1145/2934872.2959068. URL: https://doi.org/10.1145/2934872.2959068.

[21] Gilberto Bertin. "XDP in practice: integrating XDP into our DDoS mitigation pipeline". In: *Technical Conference on Linux Networking, Netdev*. 2017.

[22] Matteo Bertrone et al. "Accelerating Linux Security with EBPF Iptables". In: *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*. SIGCOMM '18. Budapest, Hungary: Association for Computing Machinery, 2018, pp. 108–110. ISBN: 9781450359153. DOI: 10.1145/3234200.3234228. URL: https://doi.org/10.1145/3234200.3234228.

[23] Monowar H Bhuyan, DK Bhattacharyya, and Jugal K Kalita. "An empirical evaluation of information metrics for low-rate and high-rate DDoS attack detection". In: *Pattern Recognition Letters* 51 (2015), pp. 1–7.

[24] Andy Bierman, Martin Björklund, and Kent Watsen. *RESTCONF Protocol*. RFC 8040. Jan. 2017. DOI: 10.17487/RFC8040. URL: https://rfc-editor.org/rfc/rfc8040.txt.

[25] Martin Bjorklund. "The YANG 1.1 Data Modeling Language". In: (2016). URL: https://tools.ietf.org/html/rfc7950.

[26] Brenden Blanco and Yunsong Lu. *Leveraging XDP for Programmable, High Performance Data Path in OpenStack*. Oct. 2016. URL: https://www.openstack.org/videos/barcelona-2016/leveraging-express-data-path-xdp-for-programmable-high-performance-data-path-in-openstack (visited on 03/17/2019).

[27] Roberto Bonafiglia et al. "Enabling NFV services on resource-constrained CPEs". In: *2016 5th IEEE International Conference on Cloud Networking (Cloudnet)*. IEEE. 2016, pp. 83–88.

[28] Gianluca Borello. *Sysdig and Falco now powered by eBPF*. Feb. 2019. URL: https://web.archive.org/web/20200528103951/https://sysdig.com/blog/sysdig-and-falco-now-powered-by-ebpf/.

[29] D. Borkmann. *net: add bpfilter*. [Online; last-retrieved 30-June-2018]. Feb. 2018. URL: https://web.archive.org/web/20190517134842/https://lwn.net/Articles/747504/.

[30] Daniel Borkmann. *bpf, x64: implement retpoline for tail call*. Feb. 2018. URL: https://web.archive.org/web/20200624134452/https://lwn.net/Articles/747859/.

[31] Daniel Borkmann. *bpf: add csum_diff helper to xdp as well*. In Linux Kernel, commit 205c380778d0. Jan. 2018. URL: https://web.archive.org/web/20200405150954/https://patchwork.ozlabs.org/patch/863867/.

[32] Daniel Borkmann. *Optimize BPF tail calls for direct jumps*. Nov. 2019. URL: https://web.archive.org/web/20200624134635/https://lwn.net/Articles/805660/.

[33] Pat Bosshart et al. "P4: Programming protocol-independent packet processors". In: *ACM SIGCOMM Computer Communication Review* 44.3 (2014), pp. 87–95.

[34] Pat Bosshart et al. "Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN". In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. SIGCOMM '13. Hong Kong, China: ACM, 2013, pp. 99–110. ISBN: 978-1-4503-2056-6. DOI: 10.1145/2486001.2486011. URL: http://doi.acm.org/10.1145/2486001.2486011.

[35] Anat Bremler-Barr, Yotam Harchol, and David Hay. "OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brazil: Association for Computing Machinery, 2016, pp. 511–524. ISBN: 9781450341936. DOI: 10.1145/2934872.2934875. URL: https://doi.org/10.1145/2934872.2934875.

[36] Jesper Dangaard Brouer. *XDP Drivers*. [Online; last-retrieved 18-September-2018]. 2018. URL: https://web.archive.org/web/20190514224949/https://prototype-kernel.readthedocs.io/en/latest/networking/XDP/implementation/drivers.html.

[37] Jesper Dangaard Brouer. *XDP redirect measurements, gotchas and tracepoints*. Aug. 2017. URL: https://web.archive.org/web/20180311113307/https://www.spinics.net/lists/xdp-newbies/msg00269.html.

[38] Jesper Dangaard Brouer. *XDP redirect memory return API*. Mar. 2018. URL: https://web.archive.org/web/20180316133406/https://lwn.net/Articles/748866/.

[39] Jesper Dangaard Brouer and Toke Høiland-Jørgensen. "XDP: challenges and future work". In: *LPC'18 Networking Track*. Linux Plumbers Conference. 2018.

[40] Maxweel Carmo et al. "Network-cloud Slicing definitions for Wi-Fi sharing systems to enhance 5G ultra dense network capabilities". In: *Wireless Communications and Mobile Computing* 2019 (2019).

[41] Adrian Caulfield, Paolo Costa, and Manya Ghobadi. "Beyond SmartNICs: Towards a Fully Programmable Cloud". In: *IEEE International Conference on High Performance Switching and Routing*. June 2018. URL: https://www.microsoft.com/en-us/research/publication/beyond-smartnics-towards-fully-programmable-cloud/.

[42] Cilium authors. *HTTP, gRPC, and Kafka Aware Network Security and Networking for Containers with BPF and XDP*. URL: https://cilium.io/.

[43] Cisco. *Cloud-native Network Function*. Jan. 2020. URL: https://web.archive.org/web/20200414130638/https://www.cisco.com/c/dam/m/en_us/network-intelligence/service-provider/digital-transformation/knowledge-network-webinars/pdfs/1128_TECHAD_CKN_PDF.pdf.

[44] Cisco. *FD.io - Vector Packet Processing*. Whitepaper. Intel, 2017. URL: https://fd.io/docs/whitepapers/FDioVPPwhitepaperJuly2017.pdf.

[45] Mike Cohen. *Monitoring Kubernetes Networking with eBPF*. Apr. 2020. URL: https://web.archive.org/web/20200624075055/https://www.flowmill.com/monitoring-kubernetes-networking-with-ebpf/.

[46] Linux community. *xdp_redirect_map_kern application*. May 2020. URL: https://web.archive.org/web/20200522140154/https://github.com/torvalds/linux/blob/master/samples/bpf/xdp_redirect_map_kern.c/.

[47] J. Corbet. *Nftables: a new packet filtering engine*. Ed. by LWN.nnet. [Online; last-retrieved 30-June-2018]. Mar. 2009. URL: https://web.archive.org/web/20200331215756/https://lwn.net/Articles/324989/.

[48] Jonathan Corbet. *BPF comes to firewalls*. [Online; last-retrieved 29-March-2019]. Feb. 2018. URL: https://web.archive.org/web/20200401061115/https://lwn.net/Articles/747551/.

[49] Jonathan Corbet. *Concurrency management in BPF*. Feb. 2019. URL: https://web.archive.org/web/20200331222010/https://lwn.net/Articles/779120/.

[50] Edward Cree. *Bounded Loops for eBPF*. Feb. 2018. URL: https://web.archive.org/web/20180308141915/https://lwn.net/Articles/748032/.

[51] *CVE-2019-7308*. Available from MITRE, CVE-ID CVE-2019-7308. Feb. 2020. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-7308.

[52] *CVE-2020-8835*. Available from MITRE, CVE-ID CVE-2020-8835. Feb. 2020. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8835.

[53] J. Daly and E. Torng. "TupleMerge: Building Online Packet Classifiers by Omitting Bits". In: *2017 26th International Conference on Computer Communication and Networks (ICCCN)*. 2017, pp. 1–10.

[54] DPDK. *Data Plane Development Kit*. June 2018. URL: https://www.dpdk.org/ (visited on 03/17/2019).

[55] DPDK. *Pktgen Traffic Generator Using DPDK*. Aug. 2018. URL: http://dpdk.org/git/apps/pktgen-dpdk.

[56] Korian Edeline et al. "mmb: flexible high-speed userspace middleboxes". In: *Proceedings of the Applied Networking Research Workshop*. 2019, pp. 62–68.

[57] Daniel E. Eisenbud et al. "Maglev: A Fast and Reliable Software Network Load Balancer". In: *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. Santa Clara, CA, 2016, pp. 523–535. URL: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/eisenbud.

[58] Paul Emmerich et al. "MoonGen: A Scriptable High-Speed Packet Generator". In: *Proceedings of the 2015 Internet Measurement Conference*. IMC '15. Tokyo, Japan: Association for Computing Machinery, 2015, pp. 275–287. ISBN: 9781450338486. DOI: 10.1145/2815675.2815692. URL: https://doi.org/10.1145/2815675.2815692.

[59] ETSI. *Network Functions Virtualization*. Feb. 2017. URL: https://web.archive.org/web/20200321091042/https://www.etsi.org/technologies/nfv.

[60] The Technology Evangelist. *Kernel Bypass = Security Bypass*. Dec. 2017. URL: https://web.archive.org/web/20200107102112/https://technologyevangelist.co/2017/12/05/kernel-bypass-security-bypass/.

[61] Arthur Fabre. *L4Drop: XDP DDoS Mitigations*. 2018. URL: https://web.archive.org/web/20190927231336/https://blog.cloudflare.com/l4drop-xdp-ebpf-based-ddos-mitigations/ (visited on 03/17/2019).

[62] Soheli Farhana et al. "Impact of big data congestion in IT: An adaptive knowledge-based Bayesian network." In: *International Journal of Electrical & Computer Engineering (2088-8708)* 10 (2020).

[63] Seyed Kaveh Fayazbakhsh et al. "Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags". In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 543–546. ISBN: 978-1-931971-09-6. URL: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/fayazbakhsh.

[64] Daniel Firestone et al. "Azure Accelerated Networking: SmartNICs in the Public Cloud". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, 2018, pp. 51–66. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/nsdi18/presentation/firestone.

[65] Matt Fleming. *A thorough introduction to eBPF*. Dec. 2017. URL: https://lwn.net/Articles/740157/ (visited on 03/17/2019).

[66] Massimo Gallo and Rafael Laufer. "ClickNF: a Modular Stack for Custom Network Functions". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, July 2018, pp. 745–757. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/atc18/presentation/gallo.

[67] Aaron Gember-Jacobson et al. "OpenNF: Enabling Innovation in Network Function Control". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM '14. Chicago, Illinois, USA: Association for Computing Machinery, 2014, pp. 163–174. ISBN: 9781450328364. DOI: 10.1145/2619239.2626313. URL: https://doi.org/10.1145/2619239.2626313.

[68] T. Graf. *Why is the kernel community replacing iptables with BPF?* [Online; last-retrieved 30-June-2018]. Apr. 2018. URL: https://web.archive.org/web/20191203111322/https://cilium.io/blog/2018/04/17/why-is-the-kernel-community-replacing-iptables/.

[69] Brendan Gregg. *Linux Extended BPF (eBPF) Tracing Tools*. May 2020. URL: https://web.archive.org/web/20200528100948/http://www.brendangregg.com/ebpf.html.

[70] Brendan Gregg. *Security Monitoring with eBPF*. Feb. 2017. URL: https://web.archive.org/web/20200523175543/http://www.brendangregg.com/Slides/BSidesSF2017_BPF_security_monitoring.pdf.

[71] Pankaj Gupta and Nick McKeown. "Packet classification using hierarchical intelligent cuttings". In: *Hot Interconnects VII*. Vol. 40. 1999.

[72] Sangjin Han et al. "PacketShader: A GPU-Accelerated Software Router". In: *Proceedings of the ACM SIGCOMM 2010 Conference*. SIGCOMM '10. New Delhi, India: Association for Computing Machinery, 2010, pp. 195–206. ISBN: 9781450302012. DOI: 10.1145/1851182.1851207. URL: https://doi.org/10.1145/1851182.1851207.

[73] Sangjin Han et al. "SoftNIC: A software NIC to augment hardware". In: (2015).

[74] Keqiang He et al. "Presto: Edge-Based Load Balancing for Fast Datacenter Networks". In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. SIGCOMM '15. London, United Kingdom: Association for Computing Machinery, 2015, pp. 465–478. ISBN: 9781450335423. DOI: 10.1145/2785956.2787507. URL: https://doi.org/10.1145/2785956.2787507.

[75] Toke Høiland-Jørgensen et al. "The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel". In: *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT '18. Heraklion, Greece: ACM, 2018, pp. 54–66. ISBN: 978-1-4503-6080-7. DOI: 10.1145/3281411.3281443. URL: http://doi.acm.org/10.1145/3281411.3281443.

[76] Christian Hopps. "Katran: A high performance layer 4 load balancer". In: https://github.com/facebookincubator/katran. Sept. 2019.

[77] Simon Horman. "TC Flower Offload". In: *Technical Conference on Linux Networking, Netdev*. 2017.

[78] Qun Huang et al. "Sketchvisor: Robust network measurement for software packet processing". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 2017, pp. 113–126.

[79] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. "NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms". In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. Seattle, WA: USENIX Association, 2014, pp. 445–458. ISBN: 9781931971096.

[80] Broadcom Inc. *Stingray PS225: Product Brief*. July 2019. URL: https://web.archive.org/web/20200625112605/https://docs.broadcom.com/doc/PS225-PB.

[81] Docker Inc. *Docker*. [Online; last-retrieved 30-June-2018]. 2018. URL: https://www.docker.com/.

[82] Google Inc. *Kubernetes: Production-Grade Container Orchestration*. [Online; last-retrieved 22-July-2019]. July 2019. URL: https://kubernetes.io/.

[83] Intel. *FD.io - Vector Packet Processing*. Whitepaper. Intel, 2017.

[84] Intel(R). *Intel® Data Direct I/O Technology*. [Online; last-retrieved 09-November-2018]. 2018. URL: https://web.archive.org/web/20170226040117/https://www.intel.it/content/www/it/it/io/data-direct-i-o-technology.html.

[85] Rishabh Iyer et al. "Performance contracts for software network functions". In: *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 2019, pp. 517–530.

[86] Ethan J. Jackson et al. "SoftFlow: A Middlebox Architecture for Open vSwitch". In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, June 2016, pp. 15–28. ISBN: 978-1-931971-30-0. URL: https://www.usenix.org/conference/atc16/technical-sessions/presentation/jackson.

[87] Ajit Jaokar. *An introduction to Cloud Native applications and Kubernetes*. Mar. 2020. URL: https://web.archive.org/web/20200413093940/https://www.datasciencecentral.com/profiles/blogs/an-introduction-to-cloud-native-applications-and-kubernetes.

[88] EunYoung Jeong et al. "mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems". In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 489–502. ISBN: 978-1-931971-09-6. URL: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong.

[89] Vimalkumar Jeyakumar et al. "EyeQ: Practical Network Performance Isolation at the Edge". In: *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX, 2013, pp. 297–311. ISBN: 978-1-931971-00-3. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/jeyakumar.

[90] Murad Kablan et al. "Stateless Network Functions: Breaking the Tight Coupling of State and Processing". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 97–112. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/kablan.

[91] József Kadlecsik and György Pásztor. "Netfilter performance testing". In: (2004).

[92] Priyanka Kamboj et al. "Detection techniques of DDoS attacks: A survey". In: *2017 4th IEEE Uttar Pradesh Section International Conference on Electrical, Computer and Electronics (UPCON)*. IEEE. 2017, pp. 675–679.

[93] Georgios P. Katsikas et al. "Metron: NFV Service Chains at the True Speed of the Underlying Hardware". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 171–186. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/nsdi18/presentation/katsikas.

[94] Naga Katta et al. "Clove: Congestion-Aware Load Balancing at the Virtual Edge". In: *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT '17. Incheon, Republic of Korea: Association for Computing Machinery, 2017, pp. 323–335. ISBN: 9781450354226. DOI: 10.1145/3143361.3143401. URL: https://doi.org/10.1145/3143361.3143401.

[95] J. Kempf, R. Austein, and IAB. *RFC3724: The Rise of the Middle and the Future of End-to-End: Reflections on the Evolution of the Internet Architecture*. USA, 2004.

[96] Eddie Kohler et al. "The Click Modular Router". In: *ACM Trans. Comput. Syst.* 18.3 (Aug. 2000), pp. 263–297. ISSN: 0734-2071. DOI: 10.1145/354871.354874. URL: https://doi.org/10.1145/354871.354874.

[97] André Kohn, Viktor Leis, and Thomas Neumann. "Adaptive execution of compiled queries". In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE. 2018, pp. 197–208.

[98] T.V. Lakshman and D. Stiliadis. "High-speed policy-based packet forwarding using efficient multi-dimensional range matching". In: *ACM SIGCOMM Computer Communication Review*. Vol. 28. 4. ACM. 1998, pp. 203–214.

[99] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE. 2004, pp. 75–86.

[100] Rafael Laufer et al. "CliMB: Enabling Network Function Composition with Click Middleboxes". In: *SIGCOMM Comput. Commun. Rev.* 46.4 (Dec. 2016), pp. 17–22. ISSN: 0146-4833. DOI: 10.1145/3027947.3027951. URL: https://doi.org/10.1145/3027947.3027951.

[101] Moon-Sang Lee. *[dpdk-dev] about poor KNI performance*. Sept. 2015. URL: https://web.archive.org/web/20200420141121/http://mails.dpdk.org/archives/dev/2015-September/023826.html.

[102] Charles E Leiserson, Harald Prokop, and Keith H Randall. "Using de Bruijn sequences to index a 1 in a computer word". In: *Available on the Internet from http://supertech. csail. mit. edu/papers. html* 3 (1998), p. 5.

[103] Ming Liu et al. "Offloading distributed applications onto smartNICs using iPipe". In: *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 318–333.

[104] Marek Majkowski. *Why we use the Linux kernel's TCP stack*. July 2016. URL: https://web.archive.org/web/20200210223048/https://blog.cloudflare.com/why-we-use-the-linux-kernels-tcp-stack/.

[105] Ilias Marinos, Robert N.M. Watson, and Mark Handley. "Network Stack Specialization for Performance". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM '14. Chicago, Illinois, USA: Association for Computing Machinery, 2014, pp. 175–186. ISBN: 9781450328364. DOI: 10.1145/2619239.2626311. URL: https://doi.org/10.1145/2619239.2626311.

[106] Joao Martins et al. "ClickOS and the Art of Network Function Virtualization". In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. Seattle, WA: USENIX Association, 2014, pp. 459–473. ISBN: 9781931971096.

[107] Steven McCanne and Van Jacobson. "The BSD Packet Filter: A New Architecture for User-level Packet Capture". In: *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*. USENIX'93. San Diego, California: USENIX Association, 1993, pp. 2–2. URL: http://dl.acm.org/citation.cfm?id=1267303.1267305.

[108] S. Miano et al. "A Service-Agnostic Software Framework for Fast and Efficient in-Kernel Network Services". In: *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 2019, pp. 1–9.

[109] S. Miano et al. "Introducing SmartNICs in Server-Based Data Plane Processing: The DDoS Mitigation Use Case". In: *IEEE Access* 7 (2019), pp. 107161–107170.

[110] Sebastiano Miano. *Custom Pktgen-DPDK version*. Oct. 2018. URL: https://github.com/sebymiano/pktgen-dpdk.

[111] Sebastiano Miano. *eBPF Iptables with Netfilter conntrack*. Mar. 2019. URL: https://github.com/sebymiano/polycube/tree/iptables_linux_conntrack.

[112] Sebastiano Miano and Fulvio Risso. "Transforming a traditional home gateway into a hardware-accelerated SDN switch". In: *International Journal of Electrical and Computer Engineering* 10.3 (2020), p. 2668.

[113] Sebastiano Miano, Fulvio Risso, and Hagen Woesner. "Partial offloading of OpenFlow rules on a traditional hardware switch ASIC". In: *2017 IEEE Conference on Network Softwarization (NetSoft)*. IEEE. 2017, pp. 1–9.

[114] Sebastiano Miano et al. "Creating complex network services with ebpf: Experience and lessons learned". In: *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. IEEE. 2018, pp. 1–8.

[115] Sebastiano Miano et al. "Securing Linux with a Faster and Scalable Iptables". In: *SIGCOMM Comput. Commun. Rev.* 49.3 (Nov. 2019), pp. 2–17. ISSN: 0146-4833. DOI: 10.1145/3371927.3371929. URL: https://doi.org/10.1145/3371927.3371929.

[116] Rui Miao et al. "SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '17. Los Angeles, CA, USA: ACM, 2017, pp. 15–28. ISBN: 978-1-4503-4653-5. DOI: 10.1145/3098822.3098824. URL: http://doi.acm.org/10.1145/3098822.3098824.

[117] Thomas Heinz Michael Bellion. *NF-HIPAC: High Performance Packet Classification for Netfilter*. Sept. 2002. URL: https://web.archive.org/web/20051129002028/http://lwn.net/Articles/10951/.

[118] Young Gyoun Moon et al. "Accelerating Flow Processing Middleboxes with Programmable NICs". In: *Proceedings of the 9th Asia-Pacific Workshop on Systems*. APSys '18. Jeju Island, Republic of Korea: ACM, 2018, 14:1–14:3. ISBN: 978-1-4503-6006-7. DOI: 10.1145/3265723.3265744. URL: http://doi.acm.org/10.1145/3265723.3265744.

[119] M. Nasimi et al. "Edge-Assisted Congestion Control Mechanism for 5G Network Using Software-Defined Networking". In: *2018 15th International Symposium on Wireless Communication Systems (ISWCS)*. 2018, pp. 1–5.

[120] Netronome. *Avoid kernel-bypass in your network infrastructure*. Jan. 2017. URL: https://web.archive.org/save/https://www.netronome.com/blog/avoid-kernel-bypass-in-your-network-infrastructure/.

[121] ntop. *PF_RING ZC (Zero Copy)*. 2018. URL: https://web.archive.org/web/20190912132122/https://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/ (visited on 03/17/2019).

[122] Orange. *Orange TV*. [Online; last-retrieved 08-April-2020]. 2020. URL: https://boutique.orange.fr/tv.

[123] Shoumik Palkar et al. "E2: A Framework for NFV Applications". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. Monterey, California: Association for Computing Machinery, 2015, pp. 121–136. ISBN: 9781450338349. DOI: 10.1145/2815400.2815423. URL: https://doi.org/10.1145/2815400.2815423.

[124] Maksim Panchenko et al. "Bolt: a practical binary optimizer for data centers and beyond". In: *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Press. 2019, pp. 2–14.

[125] Aurojit Panda et al. "NetBricks: Taking the V out of NFV". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016, pp. 203–216. ISBN: 9781931971331.

[126] Luis Pedrosa et al. "Automated Synthesis of Adversarial Workloads for Network Functions". In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '18. Budapest, Hungary: Association for Computing Machinery, 2018, pp. 372–385. ISBN: 9781450355674. DOI: 10.1145/3230543.3230573. URL: https://doi.org/10.1145/3230543.3230573.

[127] Ben Pfaff et al. "The Design and Implementation of Open vSwitch". In: *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*. NSDI'15. Oakland, CA: USENIX Association, 2015, pp. 117–130. ISBN: 978-1-931971-218.

[128] Salvatore Pontarelli et al. "FlowBlaze: Stateful Packet Processing in Hardware". In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, 2019, pp. 531–548. ISBN: 978-1-931971-49-2. URL: https://www.usenix.org/conference/nsdi19/presentation/pontarelli.

[129] Zafar Ayyub Qazi et al. "SIMPLE-Fying Middlebox Policy Enforcement Using SDN". In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. SIGCOMM '13. Hong Kong, China: Association for Computing Machinery, 2013, pp. 27–38. ISBN: 9781450320566. DOI: 10.1145/2486001.2486022. URL: https://doi.org/10.1145/2486001.2486022.

[130] Y. Qi et al. "Packet Classification Algorithms: From Theory to Practice". In: *IEEE INFOCOM 2009*. 2009, pp. 648–656.

[131] Barath Raghavan et al. "Software-defined internet architecture: decoupling architecture from infrastructure". In: *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*. 2012, pp. 43–48.

[132] Felix Rath et al. "SymPerf: Predicting Network Function Performance". In: *Proceedings of the SIGCOMM Posters and Demos*. SIGCOMM Posters and Demos '17. Los Angeles, CA, USA: Association for Computing Machinery, 2017, pp. 34–36. ISBN: 9781450350570. DOI: 10.1145/3123878.3131977. URL: https://doi.org/10.1145/3123878.3131977.

[133] Luigi Rizzo. "Netmap: a novel framework for fast packet I/O". In: *21st USENIX Security Symposium (USENIX Security 12)*. 2012, pp. 101–112.

[134] Luigi Rizzo and Giuseppe Lettieri. "VALE, a Switched Ethernet for Virtual Machines". In: *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*. CoNEXT '12. Nice, France: Association for Computing Machinery, 2012, pp. 61–72. ISBN: 9781450317757. DOI: 10.1145/2413176.2413185. URL: https://doi.org/10.1145/2413176.2413185.

[135] P. Russell. *The netfilter.org project*. [Online; last-retrieved 30-June-2018]. 1998. URL: https://netfilter.org/.

[136] Marta Rybczyńska. *Bounded loops in BPF for the 5.3 kernel*. July 2019. URL: https://web.archive.org/web/20200518033815/https://lwn.net/Articles/794934/.

[137] Telecom Italia S.p.A. *TIM Vision*. [Online; last-retrieved 08-April-2020]. 2020. URL: https://www.timvision.it/.

[138] Jerome H Saltzer, David P Reed, and David D Clark. "End-to-end arguments in system design". In: *ACM Transactions on Computer Systems (TOCS)* 2.4 (1984), pp. 277–288.

[139] Richard Sanger, Matthew Luckie, and Richard Nelson. "Towards Transforming OpenFlow Rulesets to Fit Fixed-Function Pipelines". In: *Proceedings of the Symposium on SDN Research*. SOSR '20. San Jose, CA, USA: Association for Computing Machinery, 2020, pp. 123–134. ISBN: 9781450371018. DOI: 10.1145/3373360.3380844. URL: https://doi.org/10.1145/3373360.3380844.

[140] Vyas Sekar et al. "Design and Implementation of a Consolidated Middlebox Architecture". In: *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 323–336. ISBN: 978-931971-92-8. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/sekar.

[141] Muhammad Shahbaz and Nick Feamster. "The Case for an Intermediate Representation for Programmable Data Planes". In: *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. SOSR '15. https://doi.org/10.1145/2774993.2775000. Santa Clara, California: Association for Computing Machinery, 2015. ISBN: 9781450334518. DOI: 10.1145/2774993.2775000.

[142] Muhammad Shahbaz et al. "Pisces: A programmable, protocol-independent software switch". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM. 2016, pp. 525–538.

[143] Justine Sherry et al. "Rollback-Recovery for Middleboxes". In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. SIGCOMM '15. London, United Kingdom: Association for Computing Machinery, 2015, pp. 227–240. ISBN: 9781450335423. DOI: 10.1145/2785956.2787501. URL: https://doi.org/10.1145/2785956.2787501.

[144] Sumeet Singh et al. "Packet Classification Using Multidimensional Cutting". In: *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '03. Karlsruhe, Germany: Association for Computing Machinery, 2003, pp. 213–224. ISBN: 1581137354. DOI: 10.1145/863955.863980. URL: https://doi.org/10.1145/863955.863980.

[145] Giuseppe Siracusano and Roberto Bifulco. "Is It a SmartNIC or a Key-Value Store?: Both!" In: *Proceedings of the SIGCOMM Posters and Demos*. SIGCOMM Posters and Demos '17. Los Angeles, CA, USA: ACM, 2017, pp. 138–140. ISBN: 978-1-4503-5057-0. DOI: 10.1145/3123878.3132014. URL: http://doi.acm.org/10.1145/3123878.3132014.

[146] Vibhaalakshmi Sivaraman et al. "Heavy-hitter detection entirely in the data plane". In: *Proceedings of the Symposium on SDN Research*. 2017, pp. 164–176.

[147] V. Srinivasan, S. Suri, and G. Varghese. "Packet Classification Using Tuple Space Search". In: *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM '99. Cambridge, Massachusetts, USA: Association for Computing Machinery, 1999, pp. 135–146. ISBN: 1581131356. DOI: 10.1145/316188.316216. URL: https://doi.org/10.1145/316188.316216.

[148] V. Srinivasan et al. "Fast and Scalable Layer Four Switching". In: *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM '98. Vancouver, British Columbia, Canada: Association for Computing Machinery, 1998, pp. 191–202. ISBN: 1581130031. DOI: 10.1145/285237.285282. URL: https://doi.org/10.1145/285237.285282.

[149] A. Srivastava et al. "A Recent Survey on DDoS Attacks and Defense Mechanisms". In: *Advances in Parallel Distributed Computing*. Ed. by Dhinaharan Nagamalai, Eric Renault, and Murugan Dhanuskodi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 570–580. ISBN: 978-3-642-24037-9.

[150] Alexei Starovoitov. *bpf: improve verifier scalability*. [Online; last-retrieved 02-April-2019]. Apr. 2019. URL: https://web.archive.org/web/20200406105518/https://patchwork.ozlabs.org/cover/1073775/.

[151] Alexei Starovoitov. *net: filter: rework/optimize internal BPF interpreter's instruction set*. In Linux Kernel, commit bd4cf0ed331a. Mar. 2014. URL: https://web.archive.org/web/20200406085303/https://lore.kernel.org/patchwork/patch/452162/.

[152] Alexei Starovoitov. *PATCH BPF-NEXT: Introduce BPF Spinlock*. Jan. 2019. URL: https://web.archive.org/web/20200625084026/https://lwn.net/ml/netdev/20190116050830.1881316-1-ast@kernel.org/.

[153] Radu Stoenescu et al. "SymNet: Scalable Symbolic Execution for Modern Networks". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brazil: Association for Computing Machinery, 2016, pp. 314–327. ISBN: 9781450341936. DOI: 10.1145/2934872.2934881. URL: https://doi.org/10.1145/2934872.2934881.

[154] Nick Tausanovitch. *What Makes a NIC a SmartNIC, and Why is it Needed?* Sept. 2016. URL: https://web.archive.org/web/20190616162342/https://www.netronome.com/blog/what-makes-a-nic-a-smartnic-and-why-is-it-needed/ (visited on 03/17/2019).

[155] David E Taylor and Jonathan S Turner. "Classbench: A packet classification benchmark". In: *IEEE/ACM transactions on networking* 15.3 (2007), pp. 499–511.

[156] Maroun Tork, Lina Maudlej, and Mark Silberstein. "Lynx: A SmartNIC-driven Accelerator-centric Architecture for Network Servers". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 117–131.

[157] Cheng-Chun Tu, Joe Stringer, and Justin Pettit. "Building an Extensible Open VSwitch Datapath". In: *SIGOPS Oper. Syst. Rev.* 51.1 (Sept. 2017), pp. 72–77. ISSN: 0163-5980. DOI: 10.1145/3139645.3139657. URL: https://doi.org/10.1145/3139645.3139657.

[158] William Tu. *[iovisor-dev] [PATCH RFC] bpf: add connection tracking helper functions*. [Online; last-retrieved 30-March-2019]. Sept. 2017. URL: https://web.archive.org/web/20200406105231/https://lists.linuxfoundation.org/pipermail/iovisor-dev/2017-September/001023.html.

[159] Balajee Vamanan, Gwendolyn Voskuilen, and TN Vijaykumar. "EffiCuts: optimizing packet classification for memory and throughput". In: *ACM SIGCOMM Computer Communication Review* 41.4 (2011), pp. 207–218.

[160] Nic Viljoen. *BPF, eBPF, XDP and Bpfilter...What are These Things and What do They Mean for the Enterprise?* [Online; last-retrieved 15-November-2018]. Apr. 2018. URL: https://web.archive.org/web/20190829063109/https://www.netronome.com/blog/bpf-ebpf-xdp-and-bpfilter-what-are-these-things-and-what-do-they-mean-enterprise/.

[161] J. Wallen. *An Introduction to Uncomplicated Firewall (UFW)*. Ed. by Linux.com. [Online; last-retrieved 30-June-2018]. Oct. 2015. URL: https://web.archive.org/web/20190603093744/https://www.linux.com/learn/introduction-uncomplicated-firewall-ufw.

[162] Jason Wang and David S. Miller. *XDP transmission for tuntap*. Dec. 2017. URL: https://web.archive.org/web/20180315083526/https://lwn.net/Articles/742501/.

[163] Fundamentally? What is RCU. *McKenney, Paul E. and Walpole, Jonathan*. Dec. 2007. URL: https://web.archive.org/web/20180125051005/https://lwn.net/Articles/262464/.

[164] Tong Yang et al. "Elastic Sketch: Adaptive and Fast Network-Wide Measurements". In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '18. Budapest, Hungary: Association for Computing Machinery, 2018, pp. 561–575. ISBN: 9781450355674. DOI: 10.1145/3230543.3230544. URL: https://doi.org/10.1145/3230543.3230544.

[165] Kenichi Yasukata et al. "StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs". In: *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '16. Denver, CO, USA: USENIX Association, 2016, pp. 43–56. ISBN: 9781931971300.

[166] Arseniy Zaostrovnykh et al. "A Formally Verified NAT". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '17. Los Angeles, CA, USA: Association for Computing Machinery, 2017, pp. 141–154. ISBN: 9781450346535. DOI: 10.1145/3098822.3098833. URL: https://doi.org/10.1145/3098822.3098833.

[167] Arseniy Zaostrovnykh et al. "A Formally Verified NAT". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '17. https://doi.org/10.1145/3098822.3098833. Los Angeles, CA, USA: Association for Computing Machinery, 2017, pp. 141–154. ISBN: 9781450346535. DOI: 10.1145/3098822.3098833.

[168] Rui Zhang, Saumya Debray, and Richard T Snodgrass. "Micro-specialization: dynamic code specialization of database management systems". In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. 2012, pp. 63–73.

[169] Wei Zhang et al. "OpenNetVM: A Platform for High Performance Network Service Chains". In: *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization*. HotMIddlebox '16. Florianopolis, Brazil: Association for Computing Machinery, 2016, pp. 26–31. ISBN: 9781450344241. DOI: 10.1145/2940147.2940155. URL: https://doi.org/10.1145/2940147.2940155.

[170] Yang Zhang et al. "ParaBox: Exploiting Parallelism for Virtual Network Functions in Service Chaining". In: *Proceedings of the Symposium on SDN Research*. SOSR '17. Santa Clara, CA, USA: Association for Computing Machinery, 2017, pp. 143–149. ISBN: 9781450349475. DOI: 10.1145/3050220.3050236. URL: https://doi.org/10.1145/3050220.3050236.

[171] Huapeng Zhou, Nikita, and Martin Lau. *XDP Production Usage: DDoS Protection and L4LB*. Apr. 2017. URL: https://www.netdevconf.org/2.1/slides/apr6/zhou-netdev-xdp-2017.pdf (visited on 03/17/2019).